Survey of Computer Science

Matthew Maycock

2008

Contents

Where Computers and Computer Languages Come From

(This is all very terse and meant to give you a vocabulary to follow online. Any of the information presented here can be found on wikipedia for a deeper and more thorough understanding.)

In the **1820s**, a British mathematician and engineer named **Charles Babbage** designed a machine to calculate certain values. This was his **Difference Engine**. His desire for mechanical computations was to remove the problems inherent with human calculations – namely humans and human error. He later would design a machine called the **Analytic Engine** which was a **General Computation Device**, which means that it could perform any calculation that a modern computer could. A noblewoman named **Ada Lovelace** wrote a program for the analytic engine, and was thus the world's first computer programmer. Today, the language Ada is named after her.

The analytic engine was never completed in Babbage's time, however. The next great step was completely theoretical. In the **late 1920s / early 1930s**, the famous logician **Kurt Gödel**, started tinkering in what it meant for something to be mechanically and algorithmically **computable**. He started with a set of base functions and a 'recursion' operator, a way to define computable functions with respect to the inherent recursion of the natural numbers. The group of functions defined by all of this is the set of **Primitive Recursive** functions. To this, he added a 'minimization' operator, which would, given a function as an argument, either produce a new number representing how deep of a recursion to use, or not return at all. This is equivalent to a 'while' loop (defined later). This produces the class of **General Recursive** functions.

The rest of the 30s saw the mathematicians **Alan Turing** and **Alonzo Church** defining mathematical representations of computations – the **Turing Machine** and **Lambda Calculus**, respectively. The turing machine represents a step by step algorithmic view of computation. It can be viewed as a box into which feeds an

input 'tape' with any assortment of symbols on them. Coming out of the box is an output tape upon which the machine can write an assortment of symbols (all these symbols are specific to a given turing machine – but can be corresponded with a set of numbers). Inside the box, there is an infinite tape upon which the box can write any symbol it wishes. This inside tape is the turing machine's memory (like your computer's RAM). At each 'step,' the machine knows what 'state' it is in, and can read a value off of the input tape, write a value to the output tape, and change the contents of the 'current' spot on the memory tape (as well as move the memory tape forward or backward a position).

Lambda calculus is conceptually simpler than turing machines, but can also be a bit harder to grasp. Lambda calculus is just functions, written in the form $\lambda\ x\ .\ y$. Here, the λ denotes the start of a function, the x is the parameter to the function (i.e., when you have $f(x) = x + 1$ in math class, x is the parameter). y is any collection of lambda calculus terms (including variables like x). So the function $\lambda\ x\ .\ \lambda\ y\ .\ x\ y$ is a function that takes a parameter x, and results in a function that is asking for another parameter, y. After receiving y, we apply y to x. So if we had a square function and a number 5, we could get $(\lambda\ x\ .\ \lambda\ y\ .\ x\ y)\ square\ 5$, and this would yield $(\lambda\ y\ .\ square\ y)\ 5$, which would then lead to $square\ 5$, which we all hope ends up as 25.

The part that makes lambda calculus 'weird' is that since we only ever started with functions, the parameters x and y above must be functions. Alonzo Church came up with a way to represent numbers with functions – so even the 5 and 25 above would be functions. We'll develop more of this theory later.

Mathematicians eventually conjectured that these models of computation are just as powerful as one another – there is no general recursive function that can't isn't "the same" as some turing machine and some lambda calculus term, and vice versa. The idea that all computation either equivalent or weaker than these is called the **Church-Turing Hypothesis**.

The **1940s** brought about a very real need for computers – the war effort. Computers were used to decipher Axis communications. They were also used to performed optimization computations – solving the problem of how to best allocate supplies and troops. This problem, finding an optimal solution for this specific style of problem, was the primary use of computers for most of the last century. It was projected in the 1970s that 70% of all computer time up until that point was spent solving these problems (as they are very profitable questions to answer for businesses).

The **1950s** brought about programming languages as a means for humans to be able to effeciently communicate their ideas to computers. The first two languages, **ALGOL** and **FORTRAN**, were similar in style, in a sense, to a Turing Machine. Programming them meant you were giving the computer a series of commands with

how to perform the desired functionality. The third language, **LISP**, wasn't even originally a language. It's inventor, **John McCarthy** had designed it to be a model of computation, like a Turing Machine or the Lambda Calculus. A man named **Steve Russell** realized that it itself could be a computer language, and developed it.

The **1960s & 1970s** brought about a great deal of commercial computer use. Businesses all over used computers for optimization problems (as mentioned above with the soldiers, etc...) along with automated record keeping and accounting. A plethora of programming languages and styles of programming languages emerged. While **imperative programming** had its strongholds in FORTRAN and ALGOL, **functional programming** had blossomed from LISP. As those the former were about telling a computer the steps of *how*, functional programming languages like LISP are about telling the computer *what*. This can make things much easier in some circumstances programming wise (but there is always a trade-off). We'll get to more of that later. (We bring this up now as LISP had become a used language in the 60s).

The language **C** had been developed, which was to become one of the most widely used languages, as it was the basis on which the **UNIX** operating system was developed (and was very fast and somewhat 'portable').

Also in the 60s and 70s, the new paradigm of **object oriented programming** had emerged. This model viewed the computer as a stage for different actors to interact with one another, sending messages and performing actions.

The **1980s** brought about **logic programming** in the form of **ProLog**. For years, LISP had been the poster child of the Artificial Intelligence movement, and the addition of ProLog convinced some that the world was at the dawn of a new age. It turned out, however, that ProLog was slow and not always the easiest tool to use (and AI was harder and harder than people thought). The language **C++** also became popular.

Some new languages have been developed since these times, but not much revolutionary (mainstream) work has actually resulted. We'll get into a more in depth review of some of the aforementioned languages and some others later in the reading.

Beginning Programming

The Art of Computer Programming (note – name of a famous series of books) is knowing how to describe a solution to a problem to another person in a way that a computer can mechanically execute it. This means that we need a formal way to describe things to a computer that is also easy for two separate individuals to understand. Among almost all programming languages, there are basic constructs among which you will come across that are fundamental to telling a computer what to do.

In class, we had used a "somewhat" simple pseudo-code to describe how to write computer programs instead of a specific language. From hence forth, unless otherwise noted, we shall be using the **Ruby** programming language. If you have a recent mac or linux based computer, Ruby should already be installed. To use it, navigate to the /Applications/Utilities folder on a Mac and open the Terminal application. On a linux machine, open a xterm (i.e. get to a command line). Typing `ruby mycode.rb` will have ruby execute the program in a file called mycode.rb. On windows, install the ruby for windows application you can find online, and run ruby through a dos shell (or use cygwin).

We can also get to an "interactive" version of ruby that will let us toy around with things. This is called IRB, (Interactive Ruby), and is invoked with `irb` through the Terminal / x-term / dos / cygwin program (hence forth referred to as the "command line"). You can use this for a lot of what we do throughout the book.

The simplest "program" for any language is to display a greeting message – usually the phrase Hello World. In ruby, this is accomplished with the line:

```
puts "Hello World"
```

Which will tell the ruby interpreter to put the string (puts means put string) of characters "Hello World" on the output (which is the command line window that you invoke the Ruby interpreter through anyway). If you're not using IRB, then you'll have to save this in a file and use the `ruby mycode.rb` method (there is another way, with the -e switch – if you're command line savy, try `ruby -help`).

Values, Variables, & Expressions

All (useful) programming languages have a way to represent numbers and letters. They are usually called **literals** as they literally represent what they are (i.e. 3 *is* the number Three – literally). Ruby does this in the obvious fashion. 3 represents the number three. 1.25 is one and a quarter. "ABC" is the first three letters of the alphabet, together as one entity.

With numbers comes the need to compare them. Ruby provides us with several operators for this: $<$ and $<=$ are less than and less than or equal to, respectively. $>$ and $>=$ are greater than and greater than or equal to, respectively. $==$ (two equality signs) is "equals". This is common in many programming languages as the single equal sign $=$ is used for a different operation assignment, discussed in a bit. Changing the first equal sign to an exclamation point, $!=$, results in a "not-equal" comparison... i.e. it is the opposite of an equality test. The exclamation point by itself means not. So if a comparison is true, adding an exclamation point in front of it makes it false. i.e. our test $4 < 5$ returns true, but $!(4 < 5)$ returns false.

Math Symbol	Ruby	Meaning
$<$	$<$	Less
$\leq$	$<=$	Less / Equal
$>$	$>$	Greater Than
$\geq$	$>=$	Greater / Equal
$=$	$==$	Equal
$\neq$	$!=$	Not Equal
$\neg$	!	Negation

Our numbers also let us perform the normal arithmetic operations on them. 1 + 2 will equal 3. 6 - 8 will be -2. 5 * 4 is 20. However, when we get to 1 / 2, we will see that Ruby will give us 0. This is because Ruby, and many computer languages, make a distinction between integer division and real division. To get around the issue, we can change either 1 or 2 to an explicitly real number – either 1.0 / 2 or 1 / 2.0... or both with 1.0 / 2.0.

There's another crazy operator called the **modulus** operator, shown in ruby with the % sign. Given $25 \% 4$, we see that the result is 1. What the modulus does is try to force the first number (25) into the range between 0 and right below the second number, (4-1, or 3). So for $m \% n$, it tries to place m between 0 and n - 1. For positive m, the result is the remainder of m divided by n. For negative m, imagine that n is repeatedly added to m until it is within the range.

When you're programming, you don't always want to have to use numbers and strings *everywhere* in your code. Sometimes, you'll want to "store" them for later

use without having to remember that the number you used was a 4 or a 5. This is called a **variable**. Say your program is counting the number of times something happens during its execution. At the beginning of your code (in Ruby), you would do something like this:

```
count = 0
```

Here count is our variable. It is the name for the computer to remember where you put some data. Right now, it holds the number zero. In Ruby, variables don't have **types** – i.e. count isn't a number or a string or an egg sandwich. Variables do however have values in Ruby, and values have a type. 0 is an number, and specific things can be done with integers (compared with other numbers, etc).

The = symbol above, while named *equals sign* in our workaday world, is called **assignment operator** in Ruby, and should be referred to as such. After this statement, the variable count will have the value 0. If we had had two equals signs, we would be asking for whether count was equal to 0, which is much different than telling the computer that count *is* 0.

When we come across the event we're wishing to count, we will want to add one to the variable count (this operation is called increment). It would look as so:

```
count = count + 1
```

When we have literals, variables, and the various operations that work on them, they combine to form **expressions**. An expression is anything that results in a value. So adding two numbers is an expression. Combining the separate strings "Hello", " ", and "World" into the string "Hello World" is an expression, as "Hello World" is the value it resulted in. (Strings can be added together with the + sign – magic!).

Branching & Looping

A fundamental thing to ask a computer to do is to decide to do one thing or another. This is called a **branching operation** or **if-then-else expression**. In Ruby, this looks like:

```
if expression then
  do_something
else
  do_something_else
end
```

Here, *expression* is going to be just about anything (Ruby is easy like that). In some languages, that won't fly – but we'll get to that later.

The important thing to note is that only one of the two options will be executed. Either *do_something* or *do_something_else* – not both of them, nor neither of them. Ruby also lets us drop the else part, sort of letting us just try something if the mood is right, but not do anything otherwise (this could also be achieved by leaving nothing between the else and the end).
An example:

```
if age >= 21 then
  puts "Have a drink at our wonderful bar..."
else
  puts "Please come back in a few years."
end
```

and another:

```
if age >= 21 then
  puts "What can I get you to drink?"
end
```

The next operation we need is to repeat some code a specified number of times. Say we want to write Hello World ten times. Instead of having ten puts statements, we can use a loop. In Ruby, we can "ask" a number to do the work for us (why this is so we'll get into later). This is what our new Hello World program would look like:

```
10.times {
  puts "Hello World"
}
```

– or alternatively, this syntax works as well:

```
10.times do
  puts "Hello World"
end
```

The above is what other languages would call a **for loop**. It is a loop that is executed a specified number of iterations. There are more complicated versions (ones that jump around by two or three or four; go up, down; etc), but this is good for now.

After a for loop, we come to a **while loop**. This is a loop that "doesn't know" how many times to loop. It just keeps looping until something happens in the program that tells it to stop. Assume we have some variable *number*, and we want to print out its first factor. Well, we can do that with a program like this:

```
keepLooping = true
```

```
factor = 2
while keepLooping
  if number % factor == 0 then
    puts "The first factor is #{factor}"
    keepLooping = false
  end
  factor = factor + 1
end
```

The new wonder we've run into is string interpolation. We somehow magically put the variable factor in our string using #{}. In Ruby, when this shows up in a string, the interpreter looks inside the curly braces, evaluates the expression there, and 'prints' the result into the string. In our example, that means it is placing the value of factor into the string, which will be printed by the puts function.

Back to our loop, we see that it may execute once for an even number (our first factor is 2), or it might go through every integer up to number if it is prime. We wouldn't know how many times to loop this using a for loop, so we used a while loop.

The downside of a while loop is you don't always know how many times your loop is going to need to execute. It could be infinity! This is (usually) a bad thing, as a program that enters an infinite loop doesn't get a chance to do much else. The technical term for this is **non-termination**.

Functions

Being able to repeat the same operations in the same part of your code is important, but what if there are several different parts of your program that do the same thing? We can to turn to **functions** for the solution. A function is a little chunk of code that doesn't execute on its own. Instead, some other instruction *invokes* the function, or asks it to "do it's thing." Say we wanted to have something that printed our "Hello World" greeting at the drop of a hat. In Ruby, we could have:

```
def greeting
  puts "Hello World"
end
```

Now, whenever our code sees the word greeting, it will print "Hello World" instead. It lets us replace a small word with a larger definition.

One of the advantages of functions, however, is that they have input and output – just like mathematical functions. Say we wanted to square a number, we could write the function like so

```
def square(someNumber)
  return (someNumber * someNumber)
end
```

Our square function accepts an input called someNumber. This input is called a **parameter** to the function. The function then multiplies someNumber by itself, and *returns* the result. This means that we can do something like this:

```
twentyFive = square(5)
```

While someNumber was a parameter in the function – 5, in the above, is an **argument** to the function. Arguments replace parameters in function invocations.

The return keyword tells our function to exit the function, and *replace* the actual invocation with the value that's being returned. Nothing in the function after the return will be executed, and technically, the return isn't required – Ruby will return the value of the last expression executed in a function.

Arrays & Constructs

Again, Ruby has a feature most programming languages have: **Arrays**. Arrays are sequential values that can be be indexed by a number. Say we had 20 students in a class, we could have an array *ages* with 20 different numbers in, each one representing a students age. This would allow us to access the values in the array using a loop, which would obviously be easier than writing 20 different variable names into an expression. Here's what calculating the mean age would look like:

```
total = 0.0
index = 0
20.times do
  total = total + ages[index]
  index = index + 1
end
mean = total / 20
```

In this example, we see that we can 'access' the array values using the square brackets. *ages* is the array value, and *index* determines which value in the array we're looking for. Also note that the indices start from 0, not 1. This is because the 'index' is really an 'offset' – i.e. how 'offset' from the start of the array we want to be.

There's some magical ruby syntax we can add to make the above code a bit nicer, which is:

```
total = 0.0
```

```
ages.length.times do |index|
  total = total + ages[index]
end
mean = total / ages.length
```

Here, we see that ages, as an array, has a little property called length which will tell us how many values are in the array. What we're really doing is *asking* the ages value to tell us its length, and then asking *that* value to do something so many times. The next oddity is that we have index automatically introduced 'inside' our times generated for loop. Here, index is a parameter to the loop just as someNumber was a parameter to our square function. Ruby takes care of all the magic in the background and makes sure that index starts at 0, and goes up so many times toward length. Since we no longer have to worry about whether ages has 20 or 30 values in it, this is clearly a better solution.

Declaring arrays is easy, by the way. You just have to use the square brackets again. Here's an empty array declaration, and an array with three elements:

```
empty = []
arr3 = [1, 2, 3]
```

If you we wanted to add a 4th element to our arr3, we could do it this way:

```
arr3[3] = 4
```

Since the indexing started at 0, our fourth element is the value with index 3, which is why we have arr3[3] and not arr3[4].

Another aggregate type is a **structure**. A structure is a single value that references many other values by a name. The common example is that of a student – which is a person who has a name, age, grade, and gpa. Having things collected into one value makes it easy to pass things around and only worry about the fact that we're passing a student around in our code (pass around means handing off to functions, ...). In most languages, this is how one would collect values of different types (strings, numbers, magic pancakes). Ruby, however, allows an Array to have values of any type. So structures here are purely for the organizational aspect.

Ruby structures are made by structure factories, which are created by you, the programmer. They are created using the structure factory factory called Struct, displayed in the following code.

```
Student = Struct.new("Student", "name", "age", "grade")
zach = Student.new("Zach", 23, "alumni")
kate = Student.new("Katherine", 22, 12)

zach.age = 30
```

Recursion

Sometimes a problem can be solved in a more easily and more readily manner by initially solving a smaller version of the same problem. Two common examples are computing the factorial of a number and raising a number to an exponent (we assume a non-negative integer). First, we'll see the looping versions, followed by the recursive versions.

Factorial: The factorial of a number n is the first n positive numbers multiplied together.

```
# Loop Version
def factorial(n)
  fact = 1
  n.times do |i|
    fact = fact * (i + 1)
  end
  return fact
end

# Recursive Version
def factorial(n)
  if n <= 0
    return 1
  else
    return (n * factorial(n - 1))
  end
end
```

Power: Raise a given base to a given exponent.

```
# Loop Version
def power(base, exp)
  value = 1
  exp.times do
    value = value * base
  end
  return value
end

# Recursive Version
def power(base, exp)
```

```
  if exp == 0
    return 1
  end
  value = power(base * base, exp / 2)
  if exp % 2 == 1 then
    value = value * base
  end
  return value
end
```

(Sorry about the poor styling above, the / character breaks the code typesetting facilities on my computer)

The recursive factorial seems to be just as big as the looping version, but we can take note that our if then else statement is actually an if then else expression, and rewrite it as:

```
# Small Recursive Version
def factorial(n)
  if n <= 0 then 1 else n * factorial(n - 1) end
end
```

An now we see that it is actually much simpler. We can even make it shorter by using the **ternary operator**. It lets us do an if-then-else like operation but in a less verbose fashion by using ? and : instead...

```
# Smaller Recursive Version
def factorial(n)
  n <= 0 ? 1 : n * factorial(n - 1)
end
```

The recursive power function seems to be way more complicated than the simple looping one. And it is. But there's a hidden benefit. Say we were raising some number to the 1000th power (your computer probably couldn't handle a number that big), then the first version would perform 1000 multiplications to get to that number. The recursive version takes advantage of the fact that $a^{(b+c)} = a^b * a^c$, and splits the problem in two, and then squares the result. For an exponent of 1000, this would result in probably around 20 or so multiplications, far fewer than 1000 (actually, around 14, yet an exponent of 1024 would only require 10 multiplications).

Note that each recursive definition of a function has what's called a *base case*, or what to do instead of recursion once some condition is met. In the factorial, it is when n is less than or equal to zero, and in the power function, it is when

the exponent is zero. Without this, the computer wouldn't know when to stop recursing, and your program would be stuck in an infinite loop called bottomless recursion, never to return.

Problem: Linked List

To really become a good programmer, you must actually write programs. Here, we will develop a linked list library. Ruby already has linked list functionality with its arrays – but we wouldn't learn anything using them, so we'll develop our own and learn something on the way.

The first thing we need to do is define what a linked list is. A linked list is either an empty list (in this case, we'll use the special Ruby value **nil** to represent the empty list), or it is a value followed by a linked list. This means that our list is either empty, or is a *node* followed by some other nodes (possibly none), all of it ending in nil, the empty list. Here's how we define our linked list structure:

```
LinkedList = Struct.new("LinkedList", "value", "next")
```

Now, LinkedList is a structure factory that makes linked list *nodes* – which are individual parts of the list having a value and a reference to the rest of the list. We'll develop two functions together, one to prepend and one to append onto the list, and then you can try and go fill out the rest of the library yourself (ask for help if you get to a point you are having more than a few minutes of trouble with).

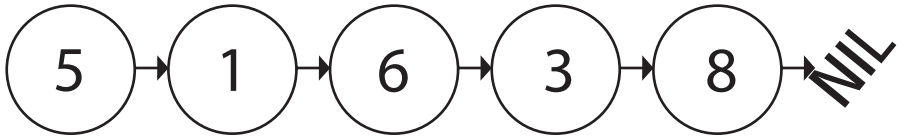

Figure 1: A Linked List

```
def prepend(list, item)
  node = LinkedList.new
  node.value = item
  node.next = list
  return node
end
```

We could have just passed the parameters item and list to the invocation of LinkedList.new, but writing it out this way seems to be a bit easier, so we'll do that for now.

```
def append(list, item)
  if list == nil then
    node = LinkedList.new
    node.value = item
    node.next = nil
    return node
  else
    list.next = append(list.next, item)
    return list
  end
end
```

The functions you should write are the follow. The notation below means that we have a function shift that takes an argument list and returns an value we're calling remaining-list. The relation between list and remaining-list (i.e. input and output) will always be dictated by the documentation, not the names chosen... but suchs names should reflect of what is happening. The same holds for the functions following the shift function.

shift(list) ⇒ [item, remaining-list]

On the empty list, this does nothing and returns the empty list. For any other list, this removes the first node, and returns the value held in that node and the rest of the list in an array.

pop(list) ⇒ [item, altered-list]

On the empty list, this does nothing and returns the empty list. For any other list, this removes the last node, and returns the value held in that node and the altered list in an array.

reverse(list) ⇒ reversed-list

Returns the list, but reversed.

nth(list, n) ⇒ item

Without altering the list, returns the n'th (first, second, third...) item held in the list.

swap(list, i, j) ⇒ list

This should swap the nodes (not the values, the actual nodes) in the ith and jth spots on the list. i.e. if i was 0 (first spot) and j was 1 (second spot), then we would make the first node point to the third node, and the second node point to the first node with their *next* values.

length(list) ⇒ number-of-nodes
This should count the number of nodes in the list.

insert(list, item, i) ⇒ list
This should alter the list by making a new node for item, and placing it at the ith position in the list. So if we had a list that was equivalent to the array [1, 2, 3, 4], inserting a 5 at position 2 would make it [1, 2, 5, 3, 4].

bubbleSort(list) ⇒ sorted-list
This should perform a **bubble sort** on the list. This should perform a bubble sort on the list: go down the list, and find the node containing the smallest element, and make that node the first node, and then sort the rest of the list.

mergeSort(list) ⇒ sorted-list
Merge sort is a more complicated sorting function, but much more efficient than bubble sort. What you have to do here is split the list into two evenly sized lists (or one bigger if it is odd in length), and then merge sort each of those lists, and then combine the two results into one sorted list.

Remember: You don't have to do everything in one function. It may be easier to use small helper functions with your solutions.

Good luck!

Formal Languages & Automata

Programming Algorithms & Data Structures

Computability Theory

Appendix A

Set Theory

The theory of sets is the basis for an informal discussion of mathematical concepts. Here, informal means that we aren't using a step by step logical proof where we justify each step when we can see that it is obvious. The everyday proofs you've seen throughout your life are informal. No one would want to read through a formal proof.

A **set** is an aggregation of items. A collection, shopping cart, filing cabinet. Anything that is a bunch of things, stuffs, and whatnots. The set of all Americans is a set, whose members are those individuals that are in some way American. The set of all sets with only two members is a set. Just about anything can be a set.

What matters with a set is what is in it. The sets $\{3, 8, 2\}$, $\{3, 2, 8\}$ and $\{8, 3, 3, 2\}$ are all the same set. They all contain the numbers 2, 3, and 8, and nothing else. Thus, sets are defined not by what order their elements are listed in, nor how many times their elements are 'placed' inside them, but only by what their elements are.

The primitive operation on a set is the test for **membership**, usually written using the symbol $\in$, as in $0 \in \{-1, 0, 1\}$. The opposite symbol, $\notin$ means that the item was not an element of the set, such as $2 \notin \{-1, 0, 1\}$.

When one set contains all the elements of another, we say that it is a subset. This can be seen with the sets $A = \{2, 4, 6\}$ and $B = \{1, 2, 3, 4, 5, 6\}$. Set A is a subset of set B, written as $A \subset B$. When we know that all the elements of a set M are in a set N, we write $M \subseteq N$, which denotes that we don't know if the two sets are equal, but we do know that M is at least a subset of N. This is the set equivalent of $<$ and $\leq$.

The **union** of two sets is the set formed by both their elements put together. If A is the set of odd integers, and B is the set of even integers, then their union – written as $A \cup B$ – is the set of all integers. Conversely, if we want to only have

elements common to both sets, which is called the **intersection**, we would write $A \cap B$. The intersection of the even and odd integers is the null set, written as $\emptyset$, as no integer is both even and odd.

If, given two sets A and B, we want to take everything that is in B out of A, we write $A \setminus B$ – **set subtraction**.

When one wants the "opposite" of a given set, or everything not in that set, it is written as A', *A prime*. This only makes sense in the context of a **universal set**, and is called the **complement** of a set. If the universal set is X, then A' is $X \setminus A$, or the set of all things not in the set A.

Appendix B

Modular Arithmetic

When you use a modulo operator (% in Ruby), you are asking for the smallest natural number 'equivalent' to a 'base' number. Here, the base number is the number on the right, and equivalent means that we want the number we have to add to the base to make it only several copies off from the 'source' number. In mathematics, this means $a \% n == b$ is the same as saying that there exists a k such that $a == n * k + b$. We read $a \% n == b$ as "a is equivalent to b, modulo n." Another way to look at the $a == n * k + b$ equation is $a - b == n * k$ – i.e. a and b are equivalent modulo n if and only if $a - b$ is a multiple of n.

When working with positive numbers, the answer of a modulo n is the remainder of a divided by n. With negative numbers, however, this is not the case. $-1 \% 5 == 4$, as one can see that if we add one copy of 5 to -1, we get 4.

Modular numbers form what is called an equivalence class – i.e. they partition the natural numbers. Here, partition means that no number is in more than one 'part' of the partition. So, if a is equivalent to b modulo n, and b is equivalent to c modulo n, then a is equivalent to c mod n, as b can't be in two parts of the partition. Also, if a is equivalent to b modulo n, then b is equivalent to a modulo n (i.e. the relationship is symmetric). When we write $[a]$ for a number a, we will mean the entire equivalence class modulo some n (if n is not mentioned, assume it is some arbitrary n).

Arithmetic on modular numbers carries out normally. $[a] + [b] == [a + b]$. To see this, note that the number a is a member of $[a]$, and like wise with b and $a + b$. Now, we can assume that a and b both have their forms of $a == i * n + k$ and $b == j * n + l$. If we add them together, we get $i * n + k + j * n + l$, which is $(i + j) * n + l + k$, which means that they are equivalent to n mod $[k + l]$. Now, because a is $i * n + k$, we see that a is equivalent to k modulo n – similarly with b and l. This means that $[a] == [k]$ and $[b] == [l]$, or that $[a] + [b] = [k] + [l]$, and

as just saw, $a+b$ is equivalent to $l+k$ modulo n. Thus, we have $[a]+[b]=[a+b]$. Since a or b could be negative, automatically see that subtraction also works with modulo arithmetic.

Next, we see that multiplication also carries over – $[a]*[b]=[a*b]$. Again, we take $a=i*n+k$ and $b=j*n+l$. $a*b=(i*n+k)*(j*n+l)$, which equals $i*j*n^2+i*l*n+j*k*n+l*k$. This can be rewritten as $(i*j*n+i*l+j*k)*n+l*k$, which leads us to see that $a*b$ is equivalent to $l*k$ modulo n (as $l*k$ is what's left after all those copies of n). As above, we can replace $[a]$ with $[k]$ and $[b]$ with $[l]$, and we come to our conclusion that $[a]*[b]==[l]*[k]==[l*k]==[a*b]$.

Appendix C

How To Program

While earlier chapters described the syntax of Ruby and constructs of most programming languages, they did not tell you, the reader, actually how to turn that into a working computer program. Programming, as with most things in life, is a series of steps that are broken down into a series of steps, and so on and so on. The following should be a helpful guide to the process, with an example following.

Step One: What are you doing? This seems like an obvious step, but it is important to stand back and actually look at what your goal is. Do you want a program that will perform some internet task? One that will provide an interface to the user and perform a scientific calculation? Writing this down in general yet specific terms can be helpful. What this oxymoronic statement means is that you should write a "bird's eye view" of what you want your program to do, but have what you write be specific to the problem at hand.

Step Two: From where to where? Figure out what input you're going to need and what the output requirements are. Both of these might be decided by someone else and you might just be the hands that make it all happen, or it could be all up to you. Knowing your end points will give your proposed solution a definite space to live in. You'll also know more of what has to happen than in the last step.

Step Three: Giant steps... Decide how you want to handle the problem in general steps. This means don't worry about what variables you're going to come up with to hold things – but do decide if you're going to have to sort a list, search a flat file database, or perform some crazy image manipulations.

Step Four: Diagram it! You'll definitely want to make diagrams about the flow of information from the previous step. This will help you understand how input will map to output and maybe find some pitfalls with your logic before you have to write any code (where errors would be harder to find).

Step Five: Recursion is important. For each of the giant steps decided in step three, you'll now reapply this process starting at step one. Use this to help you decide what should be its own function and what shouldn't be.

Using this process, you'll have a good road map for what to do to make your program a success. But this isn't the only thing you'll have to worry about. Another part of knowing how to program is code literacy. You're going to want to learn how to read other people's code – a task which will require you to keep a mental model of variables and domain specific knowledge in your head. You'll have to be able to diagram someone else's code, like in step four above, so that you can better understand it. Don't always count on there being documentation with or about the code you're going to be looking at, and remember that unless you're writing in pure assembly, you're always using someone else's code.

Appendix D

Lambda Calculus

Lambda calculus was briefly defined in the first chapter. Here, we shall examine it more, for those who wish to have a greater understanding of it (as it is hip to do lately, as functional languages become more hip). Lambda calculus consists of terms. A term can be one of the following items:

- Variable: This is exactly what it sounds like. It is some place holder for a value. Note that variables here don't vary. What varies is what they can be, not what they are.
- Application: If we have a function f and term x, then fx is the application of f to the term x. If f were $square$ the square function, and x were 2, then the result would be 4.
- Abstraction: We abstract out a variable from a term and allow it to be set as a parameter. In the term xyz, we can write $\lambda\ x\ .\ x\ y\ z$, and now we have a function where x is the parameter – so the term $(\lambda\ x\ .\ x\ y\ z)\ g$ results in $g\ y\ z$ when x is replaced by g. It is common to abbreviate $\lambda\ x\ .\ \lambda\ y\ .\ z$ as $\lambda\ x\ y\ .\ z$.

This is our basic definition. Now, with this, we want to start defining computations – i.e. things to do. One thing we need to be able to do is to use numbers; and we use numbers to count. So let's make a representation of numbers that does counting, or more generally, does anything a set number of times:

To define the number $zero$, we will take a function f, and a parameter x, and apply f to x zero times. Thus, we have the lambda term $\lambda\ f\ x\ .\ x$. Now, to represent one, we want to apply f to x once, which is $\lambda\ f\ x\ .\ f\ x$. In general, given a number n, the successor of n is given by $\lambda\ n\ f\ x\ .\ f\ (n\ x)$. We can also define addition, without repeatedly using the successor function, by instead noting that addition of

m and n in our representation is just performing a function m times and n times: $\lambda\ m\ n\ f\ x\ .\ m\ f\ (n\ f\ x)$, which will "first" apply f n times to x, and then apply f m times to the result. Multiplication is just as easy: given m and n, we want to do m operations n times (or vice versa). This yields $\lambda\ m\ n\ f\ x\ .\ m\ (n\ f)\ x$.

Taking the predecessor of a number, however, is a bit more difficult. We will note that one possible definition is $\lambda\ n\ f\ x\ .\ n\ (\lambda\ g\ h\ .\ h\ (g\ f))\ (\lambda\ u\ .\ x)\ (\lambda\ u\ .\ u)$. This is a special predecessor function, though, as it gives the predecessor of $zero$ as $zero$, and every other number n as $n - 1$. With this, or any other version of the function, subtraction is trivially: $\lambda\ m\ n\ .\ n\ pred\ m$, where pred is the predecessor function.

One important thing to be able to do with numbers is compare them. We'll describe how to determine whether a number is $zero$. First, we'll take it as a given that we have some value to use as $true$ and one to use as $false$. Later, we'll describe what these are and how they're used – but needless to say they are the same true and false as you're used to (for the most part).

So the function we're going to make is called $zerop$: the zero predicate. It will return $true$ if a given number is $zero$, and $false$ if it is larger than $zero$ (no guarantee of results when used on non-numbers). Since $zero$ is $\lambda\ f\ x\ .\ x$, and we want it to result in $true$, we must have that our x is $true$. Thus, our $zerop$ function has one piece complete: $zerop := \lambda\ n\ .\ n\ ?\ true$, where we don't know what to put for ? yet. Now, every non-zero number has the form of $\lambda\ f\ x\ .\ f^i\ x$ for some i – i.e. we're applying f i times to x. Thus, we can see that it is also the same as $\lambda\ f\ x\ .\ f\ (f^{i-1}\ x)$. For our test, that gives us some idea of what f needs to be – it needs to return $false$ when applied to anything, as it will only come into play with numbers greater than 0. So, f is the constant function that returns $false$. We define the constant function, called K, as $\lambda\ x\ y\ .\ x$, and it's dual K^* as $\lambda\ x\ y\ .\ y$. Thus, our $zerop$ function is $\lambda\ n\ .\ n\ (K\ false)\ true$.

Problem: Define the functions $\leq$, $<$, $=$, $>$, and $\geq$.

Next, we come to terms with the fact that we have to actually define $true$ and $false$. Since these two are used to determine what to do in an if expression of most languages, we should see what that means. We will usually have the expression **if** *boolean* **then** *true-expression* **else** *false-expression*. If we ignore syntax, we notice that we only have three 'variables' here: boolean, true-expression, and false-expression. When boolean is $true$, we return true-expression and throw away false-expression. When boolean is $false$, return false-expression and throw away true-expression. Thus, we get the following: $true$:= K, $false$:= K^* (thus making our $zerop$ function $\lambda\ n\ .\ n\ (K\ K^*)\ K$). We can use these themselves as if expressions. Instead of having syntax, our boolean values will just return the correct value. So, if we have a $<$ function, instead of writing **if** $a < b$ **then** $a + 1$ **else** $b + 2$, we have $(a < b)\ (+\ a\ 1)\ (+\ b\ 2)$.

Problem: Define the operators AND, OR, XOR, and NOT.

Given that our numbers themselves perform operations a set number of times (for loops), and we have if expressions through variables, we find ourselves missing one key item: while loops! We can achieve these through the wonders of recursion. But we don't have recursion you say? Well, check this here Y combinator out: $\lambda\, f\, .\, (\lambda\, x\, .\, f\, (x\, x))\, (\lambda\, x\, .\, f\, (x\, x))$. With this nifty doo-dad, we can get recursion. (The key is that the bound function f is used inside a term that keeps feeding itself to itself, reaccepting input over and over until the user decides not to go any further). If you don't believe this works, write your own factorial function with it.

The while loop can be described with three simple components: an initial information state – this is what the initial 'world' looks like to the while loop. A predicate test – this is a function that looks at the current world state and returns either a $true$ or a $false$. A body – this takes the current world and transforms it into a new world. Our while loop looks like the following:

$Y\ (\lambda\ while\ predicate\ body\ init\ .$
$\quad (predicate\ init)$
$\quad (while\ predicate\ body\ (body\ init))$
$\quad init)$

Here, $(predicate\ init)$ should return either $true$ or $false$. If it returns $false$, then the whole expression just returns the current world state, which is $init$. Otherwise, we take $while$ (which was given to us by our magical Y combinator), and invoke it with the same $predicate$ and $body$, but use $body$ to advance $init$ to the next world state. Thus, we have a while loop.

Let's take a stab at defining a 'program' in lambda calculus. We will try to determine if a number is prime or not.

$\lambda\, n\, .$
$\quad (\lambda\, i\, .\, >\, (square\ i)\ n)\ (while$
$\quad\quad (\lambda\, i\, .\, and\ (\leq\ \ (square\ i)\ n)\ (not\ (divp\ i\ n)))$
$\quad\quad succ$
$\quad\quad 2)$

The question now is where did all these extra functions come from, and what did they do. Where these functions come from is simple: we just need to put this entire term inside a lambda abstraction and define them as parameters. So, for $square$, we can have $(\lambda\ square\ .\ our\ prime\ test)\ (\lambda\ i\ .\ *\ i\ i)$, and using * as the multiplication we defined earlier, we have a square function inside our lambda term. The other functions are their obvious values. not, and, and $\leq$ are the functions you defined in your problem sets, $succ$ is the successor function. The unfamiliar function of $divp$, however, needs to be defined by you the reader – given what you have now, it should be easy. This function should return whether the first argument evenly divides the second.

Problem: Define a function that makes it a bit easier to write numbers out. What we want are the functions *radix* and *xidar*, and they should act as follows: $radix\ r\ a_1\ a_2\ ...\ a_n\ xidar$ should return the number represented by the string $a_1a_2...a_n$ in base r.

Problem: Define a linked list type and rewrite the linked list library from before using lambda calculus.