PROGRAMMING LANGUAGE CONCEPTS

SIDDHANT CHAUDHARY

These are my course notes for the course PROGRAMMING LANGUAGE CONCEPTS that I undertook in my fourth semester.

Contents

Date: January 2021.

1. Object Oriented Programming in Java

1.1. Setting up Java.

First, download the latext version of the Java JDK from the Oracle website. Then, install the JDK. Finally, add the Java compiler to your system's PATH variable. To do this, do the following.

(1) Go to My Computer (or This PC). Right click and select properties.
(2) Then go to Advanced system settings.
(3) Then click on Environment Variables... button.
(4) Then edit the PATH variable, and add the location of the `bin` folder of your JDK installation. Typically, it is something like

```
C:\Program Files\Java\jdk-15.0.1\bin
```

1.2. Basics.

Java is based based on the idea of *object oriented programming (OOP)*, and so a Java program is a collection of classes. Each class `xyz` is in a separate file `xyz.java`. To start the computation, there must be one class, usually called `Main` which contains a *static method* (see the section 1.5 **Static Components** for more info about `static`), and this method is `main`. This is done by including this code in the `Main.java` class.

```
class Main{
    public static void main (String[] args){
        System.out.println("Hello World!"); //a Hello World program
    }
}
```

1.3. Running Programs.

Java programs are interpreted on *Java Virtual Machines (JVM)*. `javac` compiles Java into bytecode for the JVM. The command `javac xyz.java` on the terminal creates a *class* file `xyz.class`, and this class file can be compiled using the command `java xyz` (note that there is no `.class` in this command). The good thing about the `javac` compiler is that it follows all dependencies, and it compiles all the classes required. For instance, if in our `xyz` class there is an object of type `abc`, then the command `javac xyz.java` will automatically compile `abc.java` as well.

Caution: Make sure that you are running these commands in the terminal from the location where your `xyz.java` file is located.

1.4. Classes and Objects.

In Java, the template or the blueprint used to define an object is known as a *class*. An instance of a class is known as an *object*. Let's get right into the syntax used in Java. In Java, there is a keyword `new` which is used to allocate memory for new objects. Then, properties of the objects as defined in the class are accessed using the `.` symbol. For instance, suppose we have a class `stack` already defined (which will be a blueprint for the stack data structure). Then, we can create instances of `stack` as follows.

```
stack s , t; /*References to stack*/
s = new stack; /*create one stack*/
t = new stack; /*... and another*/
```

```
    s.push(7)
```

Note that the following piece of code *does not* allocate a memory for a new instance of `stack`.

```
    s = new stack;
    t = s  /*just assigns another name*/
```

1.5. Public vs Private. The keyword `private` is used to make sure that the variables defined inside a class are not visible to external code. An example is the following.

```
    class stack{
        private int values[100];
        private int top_of_stack = 0;
    }
```

So, in this case the variables `values` and `top_of_stack` will *not* be visible to the external code.

Remark 1.0.1. There is an exception to this. In the above example, suppose we have two variables `s` and `t` of the type `stack`. Then, `s` and `t` will be able to access each other's `values` and `top_of_stack` variables. In general, objects of the *same type* can access each other's `private` variables.

1.6. Static Components. Suppose we have a class `stack` as earlier, and suppose we want to use execute a method of this class in our program. So, we make a new object of type `stack` as follows.

```
    stack s;
    s = new stack;
```

Now, the question is: where is this code written? And the answer is simple: this code is written in some other class which declares this object and will then execute a method via this object. Now, to execute a method of this class, we again have to do the same thing; define an object of this new class in another class. But this cycle will continue, and we will never be actually able to *execute* any code. This is where *static components* come into play. A *static function* is a function which can be invoked without instantiating any class. Static functions exist even if there is no object instance of any class. Infact, we can use the `static` keyword to declare fields as well. For instance, suppose we have the following class.

```
    class Math{
        public static double PI = 3.1415927;
        public static double E = 2.7182818;
    }
```

So, the variables `PI` and `E` can be used throughout the program freely, and because of the keyword `static`, they will have one fixed memory allocated for them.

Example 1.1. Here is a clever use of the `static` keyword. Suppose we have a class `stack`, and we want to keep track of the *total* number of push operations across all the stacks in our program. So, we do the following.

```
class stack{
    ...
    private static int num_push = 0; /*number of pushes across all
        stacks*/

    push (int i, ...){
        ...
        num_push++; /*update the static variable*/
        ...
    }
    ...
}
```

Caution: Make sure that any static method *does not* use any non-static variable.

1.7. **Making read-only variables.** The keyword `final` is used to declare read-only variables, i.e variables whose value cannot be changed after initialisation.

1.8. **Constructors.** Suppose we have a class `Date`. Whenever we instantiate the class `Date`, a special method called the *constructor* will be invoked. For instance, suppose we have the declaration `Date x = new Date();` in our code. At the time of creation of `x`, the *constructor* method for the `Date` class will be invoked. The constructor method has the same name as the class; in our case, the constructor method has the name `Date`. The class `Date` will look something like below.

```
class Date{
    private int day , month , year;
    public Date(int d , int m , int y){
        day = d; month = m; year = y;
    }
}
```

We can then initialise an object as follows.

```
Date d = new Date(27 , 1 , 2009);
```

Let us now look at some properties of constructors.

(1) We can have more than one constructor, and this uses the concept of *function overloading* in Java (you can read more about this in the Java docs).
(2) Here is a neat way of invoking an earlier constructor.

```
class Date{
    class Date{
        private int day, month, year;

        public Date(int d , int m , int y){
            day = d; month = m; year = y;
        }

        public Date(int d , int m){
            this(d , m , 2009); //calling the earlier
                constructor
        }
    }
}
```

(3) If no constructors are specified, then there is a *default constructor* which Java automatically creates for you.

1.9. **Strings in Java.** `String` is a built in class in Java. These are declared using double quotes as usual, i.e something like

```
String s = "Hello";
```

It must be noted that a `String` object is *not* an array. So for example in the above definition, something like `s[3]` does not make sense.

1.10. **Arrays.** Without a surprise, arrays are also objects in Java. They are typically declared as follows.

```
int [] a; //can also be written as int a[]
a = new int[100];
```

We can combine the above declaration into a single statement as follows.

```
int [] a = new int[100];
```

This warrants a question: in the above statement, we have already declared a to be an array of type `int`. Then why do we have to write `new int[100]` again? We will soon see the answer to this question.

To get the size of an array, we can simply do `a.length` (Caution: for strings, `length` is a *method*, i.e we have to say `s.length()` to get the length of a string).

Example 1.2. We have seen that the argument to the `main` method in any Java program is of type `String[] args`. Here, `args` is an array of variables of type `String`, and these arguments can be passed in the command line while running the Java program.

1.11. **Subclasses.** This is best understood with an example. So, we will have a class called `Employee`. Here is the description.

(1) The class `Employee` is used to contain employee data.

```
class Employee{
    private String name;
    private double salary;

    //Some Constructors...

    //"mutator" methods
    public boolean setName(String s){ ... }
```

```
        public boolean setSalary(double x){ ... }

        //"accessor" methods
        public String getName(){ ... }
        public double getSalary(){ ... }

        //other methods
        double bonus(float percent){
            return (percent/100.0)*salary;
        }
    }
```

(2) Managers are special type of employees with extra priveledges. We implement this as a *subclass* in Java.

```
    class Manager extends Employee{
        private String secretary;
        public boolean setSecretary(String s){ ... }
        public String getSecretary(){ ... }
        double bonus(float percent){
            return (percent/100.0)*2*salary;
        }
    }
```

(3) `Manager` inherits other fields and methods from `Employees`. For instance, every `Manager` object has a `name`, `salary` and methods to access and manipulate these.
(4) However, a `Manager` object will not automatically get access to the private data of the parent class.
(5) A subclass can override methods of its parent classes.
(6) Suppose we do the following.

```
    Employee e = new Manager(...)
```

Then, the questions is: can we invoke `e.setSecretary()`? And the answer is *no*, because of Java's static typechecking. `e` can only refer to the methods in `Employee`. However, note that both `Employee` and `Manager` have a method called `bonus`. If we invoke `e.bonus(p)`, then the `bonus` of the `Manager` class will be invoked! This is called *dynamic dispatch*.

1.12. **Multiple Inheritance.** The short story is: Java simply *does not* allow multiple inheritance. Now we see the long story. Suppose we have something like this.

```
class C1 {
    public int f(...){...}
}
class C2 {
    public int f(...){...}
}
class C3 extends C1,C2 {
    ...
}
```

The question is: what happens if we invoke f from an object of type `C3`? Clearly, there is some ambiguity here. To deal with this, Java simply does not allow this to happen, i.e any class can have *atmost one* parent.

So if we create a class inheritance graph, where there is an edge from a parent class to a child class, then the graph will be a tree: each class will have atmost one parent. Infact, in Java there is a universal superclass `Object`.

1.13. **Abstract Classes.** Suppose we have three classes `Circle`, `Square` and `Rectangle`, and we want to combine these under a common class named `Shape`. Now, we want to *enforce* every shape to define a function

```
public double perimeter()
```

which will return the perimeter of the shape. One solution to this problem is the following: define a function in `Shape` that returns an absurd value

```
class Shape{
    ...
    public double perimeter (){return -1.0;}
    ...
}
```

and then we can rely on any subclass of `Shape` to redefine this function. However this is not quite what we want, because we want *every* subclass to have a function called `perimeter`.

To solve this problem, we use the `abstract` keyword, and we provide an *abstract declaration* in `Shape`:

```
class Shape{
    ...
    public abstract double perimeter(); //no function definition here,
        just a declaration
}
```

This will *force* every subclass of `Shape` to prove a concrete implementation of this function.

Now note that once we have an abstract function defined inside the class `Shape`, the class `Shape` itself must be defined to be abstract, because we cannot make objects of type `Shape` because of the obvious fact that there is no definition for `perimeter` inside the class `Shape`. So, we have to do:

```
abstract class Shape{
    ...
    public abstract double perimeter(); //no function definition here,
        just a declaration
}
```

1.14. **Interfaces.** An *interface* is an abstract class with no concrete components, and any class that extends an interface is said to *implement* it. The good thing about interfaces is that a class can implement multiple interfaces. On the other hand, recall that a class can have *atmost one* parent class.

An example would be the following.

```
abstract class Shape{
    public abstract double perimeter();
}
interface Comparable {
    public abstract int cmp(Comparable s);
}
class Circle extends Shape implements Comparable{
    public double perimeter(){...}
    public int cmp(Comparable s){...}
}
```

1.15. **Generic Programming.** Suppose we want to implement a linked list in Java such that the list can contain objects of any data type, but the list must be *homogeneous*, i.e the list must contain objects of the same type. To do this, we can create a *template* as follows.

```
public class Node<T>{
    public T data; //T can be any data type
    public node next;
    ...
}
public class LinkedList<T>{
    private int size;
    private Node first;

    public T head(){
        T returnval = null;
        if (first != null){
            returnval = first.data;
            first = first.next();
        }
        return returnval;
    }
    public void insert (T newdata){
        ...
    }
}
```

So we have just declared a template for a linked list class that can hold any object of any type, but the type across a list must be the same. The variable `T` above is a placeholder for the data type. For example, we can do the following with the above definition.

```
LinkedList<Ticket> ticketList = new LinkedList<Ticket>();
LinkedList<Date> dateList = new LinkedList<Date>();
Ticket t = new Ticket();
Date d = new Date();
ticketlist.insert(t);
datelist.insert(d);
```

This way of programming is called *generic programming*.

We can even write generic functions in Java. For example, suppose we want to make a function which takes two arrays of the same type, and copies the first array into the other. This can be done as follows.

```
public <T> void arraycopy (T[] src , T[] tgt){ //the generic type comes
    before the return type of the function
    int i , limit;
    limit = min(src.length , tgt.length);
    for (i = 0; i < limit; i++){
        tgt[i] = src[i];
    }
}
```

2. Lambda Calculus

2.1. **Syntax.**

Suppose we have a countably infinite set of variables called Var. The set Λ of lambda expressions is given by

$$\Lambda = x \mid \lambda x.M \mid MN$$

where $x \in \mathrm{Var}$ and $M, N \in \Lambda$. The notation $\lambda x.M$ is called *function abstraction*; this just means we have a function of x with computation rule M. The notation MN is called *function application*; apply the function M to the argument N. So we can really think of the set Λ as the language generated by a context free grammar. Here is some more syntax.

(1) Function application associates to the left, i.e

$$MNP = (MN)P$$

(2) Function abstraction associates to the right, i.e

$$\lambda x.\lambda y.M = \lambda x.(\lambda y.M)$$

2.2. **The computation rule.**

The basic rule for computing is called β-*reduction* (or *contraction*).

- $(\lambda x.M)N \underset{\beta}{\to} M[x := N]$. Informally, we say the compute the expression M where $x = N$.
- A term of the form $(\lambda x.M)N$ is called *redex*, which stands for *reducible expression*.
- $M[x := N]$ is called the *contractum*. The expression $M[x := N]$ means to substitute *free* occurrences of x in M by N.

2.3. **Free and Bound Occurrences.**

An occurence of a variable x in M is *free* if it does not occur in the scope of a λx inside M. This definition is a bit ambiguous, so we will use an alternative inductive definition. Let M be a lambda expression. The set of all variables occurring free in M will be denoted by $FV(M)$ and the set of all variables occurring bound in M will be denoted by $BV(M)$.

Definition 2.1. The set $FV(M)$ is defined inductively as follows:

(1) $FV(x) = \{x\}$ for any $x \in \mathrm{Var}$.
(2) $FV(MN) = FV(M) \cup FV(N)$.
(3) $FV(\lambda x.M) = FV(M) \setminus \{x\}$.

Definition 2.2. The set $BV(M)$ is defined inductively as follows.

(1) $BV(x) = \phi$ for any $x \in \mathrm{Var}$.
(2) $BV(MN) = BV(M) \cup BV(N)$.
(3) $BV(\lambda x.M) = BV(M) \cup (\{x\} \cap FV(M))$.

Example 2.1. Consider the example $M = xy(\lambda x.z)(\lambda y.y)$. Then one checks that $FV(M) = \{x, y, z\}$ and $BV(M) = \{y\}$. This example shows that $FV(M)$ and $BV(M)$ need not be disjoint sets.

Remark 2.0.1. A good way to think about free and bound variables is as follows: consider a program written in some language. A variable that is local to some function (i.e variables denoting parameters of functions) is a bound variable, while global variables are free variables.

Remark 2.0.2. One can ensure by suitable renaming of variables (called α-*renaming*) that $FV(M) \cap BV(M) = \phi$.

2.4. **Variable Capture.** Sometimes while β-reducing, a variable may become bound in a lambda expression, and this should be avoided. For example, consider $N = \lambda x.(\lambda y.xy)$ and $M = Ny$.

(1) N takes two arguments and applies the first argument to the second.
(2) M fixes the first argument of N. M simply means to take an argument and apply y to it.

If we β-reduce M we get $N[x := y]$, which is equivalent to the expression $\lambda y.yy$. But this is not what we want: this expression means take an argument and apply it to itself.

The problem above is that the y substituted for the inner x has been confused with the y bound by λy. So, we rename the bound variables to avoid variable capture

$$(\lambda x.(\lambda y.xy))y = (\lambda x.(\lambda z.xz))y \underset{\beta}{\rightarrow} \lambda z.yz$$

It is clear that renaming the bound variables does not change the function. For instance, $\sin z$ and $\sin y$ mean the same thing.

2.5. **Applying the computation rule in context.** We can contract a redex appearing anywhere inside an expression. We have the following rules for β-reductions.

(1) $(\lambda x.M)N \underset{\beta}{\rightarrow} M[x := N]$.
(2) If $M \underset{\beta}{\rightarrow} M'$ then $MN \underset{\beta}{\rightarrow} M'N$.
(3) If $N \underset{\beta}{\rightarrow} N'$ then $MN \underset{\beta}{\rightarrow} MN'$.
(4) If $M \underset{\beta}{\rightarrow} M'$ then $\lambda x.M \underset{\beta}{\rightarrow} \lambda x.M'$.

So if we have a lambda expression M that has redexes appearing inside it, then we can β-reduce all redexes to get an expression N, i.e $M \underset{\beta}{\overset{*}{\rightarrow}} N$ via a sequence of β reductions

$$M = M_0 \underset{\beta}{\rightarrow} M_1 \underset{\beta}{\rightarrow} \cdots \underset{\beta}{\rightarrow} M_k = N$$

2.6. Encoding arithmetic in λ-calculus. First, some notation. Multiple function abstractions can be written in a more compact form. For instance, we write

$$\lambda x.\lambda y.\lambda z.M = \lambda xyz.M$$

Let f be any expression, and let $x \in \mathrm{Var}$. Define

$$\begin{aligned} f^0 x &:= x \\ f^{n+1} x &:= f(f^n x) \end{aligned}$$

We encode the natural numbers as follows. We define

$$[n] := \lambda f.\lambda x.f^n x = \lambda fx.f^n x$$

For instance,

$$\begin{aligned} [0] &= \lambda fx.x \\ [1] &= \lambda fx.fx \\ [2] &= \lambda fx.f(fx) \\ [3] &= \lambda fx.f(f(fx)) \\ &\vdots \end{aligned}$$

From this definition, it can be checked by simple β-reduction that

$$[n]gy = g^n y$$

2.7. Encoding Arithmetic Functions. In this section, we will see how to encode various arithmetic functions in λ-calculus.

2.7.1. *Successor Function.* Consider the successor function on natural numbers, i.e the function $x \mapsto x+1$ for $x \in \mathbb{N}$. We want to encode this function in the form of a lambda expression. Suppose we call it $[succ]$. We want

$$[succ][n] = [n+1] \tag{2.1}$$

for all $n \in \mathbb{N}$. Now observe that

$$[n]fx \xrightarrow{*} f^n x$$

as we saw above. This means that

$$f([n]fx) \xrightarrow{*} f(f^n x) = f^{n+1}x$$

and hence

$$\lambda fx.f([n]fx) \xrightarrow{*} \lambda fxf^{n+1}x = [n+1]$$

So, we see that

$$[succ] = \lambda pfx.f(pfx)$$

From this definition, we can easily show that equation (2.1) holds.

2.7.2. *Addition.* We want to encode the function $[plus]$ that does the following: it takes two natural numbers m and n, and it returns $m+n$. Note that doing $[plus][m][n]$ is the same as doing $[m]succ[n]$ (which β-reduces to $succ^m[n]$). So, one way of encoding this function is

$$[plus] = \lambda pq.p[succ]q$$

Another way of encoding this function is the following:

$$[plus] = \lambda pqfx.pf(qfx)$$

This works because

$$\begin{aligned} [plus][m][n] &\xrightarrow{*} \lambda fx.[m]f([n]fx) \\ &\xrightarrow{*} \lambda fx.[m]f(f^n x) \\ &\xrightarrow{*} \lambda fx.f^m(f^n x) \\ &= \lambda fx.f^{m+n}x \\ &= [m+n] \end{aligned}$$

2.7.3. *Multiplication.* One way to encode this is as follows:

$$[mult] = \lambda pqf.p(qf)$$

To see this, observe that

$$\begin{aligned} [mult][m][n] &\xrightarrow{*} [m]([n]f) \\ &\xrightarrow{*} [m](\lambda y.f^n y) \\ &= (\lambda hz.h^m z)(\lambda y.f^n y) \\ &\xrightarrow{*} \lambda z.(\lambda y.f^n y)^m z \end{aligned}$$

Now, it can be shown by induction that $\lambda z.(\lambda y.f^n y)^m z \xrightarrow{*} \lambda z.f^{mn}z$ (try to prove this yourself) and hence we see that

$$[mult][m][n] \xrightarrow{*} \lambda z.f^{mn}z = [mn]$$

2.7.4. *Exponentiation.* We want to find a lambda expression for the function exp which gives $exp(m,n) = n^m$. A simple way to do this is

$$[exp] = \lambda pq.pq$$

Complete the proof of why this works!

3. Computability in Lambda Calculus

3.1. **Recursive Functions.** Let us begin with a definition. For a vector $(n_1, \cdots, n_k) \in \mathbb{N}^k$, we will use the notation $\vec{n}$.

Definition 3.1. $f : \mathbb{N}^k \to \mathbb{N}$ is obtained by *composition* from $g : \mathbb{N}^l \to \mathbb{N}$ and $h_1, ..., h_l : \mathbb{N}^k \to \mathbb{N}$ if

$$f(\vec{n}) = g(h_1(\vec{n}), \cdots, h_l(\vec{n}))$$

In this case, we use the notation $f = g \circ (h_1, \cdots, h_l)$.

Definition 3.2. $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is obtained by *primitive recursion* from $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ if

$$f(0, \vec{n}) = g(\vec{n})$$
$$f(i + 1, \vec{n}) = h(i, f(i, \vec{n}), \vec{n})$$

This definition is equivalent to the following for loop.

```
result = g(n1,...,nk); //f(0,n1,...,nk)
for (i = 0; i < n; i++){
    //computing f(i + 1,n1,...,nk)
    result = h(i , result , n1,...,nk);
}
return result;
```

Note that if both g, h are total functions, then f is also a total function.

Definition 3.3. $f : \mathbb{N}^k \to \mathbb{N}$ is obtained by *μ-recursion* or *minimization* from $g : \mathbb{N}^{k+1} \to \mathbb{N}$ if

$$f(\vec{n}) = \begin{cases} i & \text{if } g(i, \vec{n}) = 0 \text{ and } \forall\ j < i\ :\ g(j, \vec{n}) > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

The notation we use is $f(\vec{n}) = \mu i(g(i, \vec{n}) = 0)$. Here μ stands for *minimal*. Here note that even if g is a total function, f need not be total. This definition is equivalent to a while loop.

```
i = 0;
while (g(i,n1,...,nk) > 0){
    i = i + 1;
}
return i;
```

Definition 3.4. The class of *primitive recursive function* is the smallest class of functions for which the following hold.

(1) This class contains all the initial functions:
  (a) Zero $Z(n) = 0$
  (b) Successor $S(n) = n + 1$
  (c) Projection $\pi_i^k(n_1, ..., n_k) = n_i$
(2) This class is closed under composition and primitive recursion.

Definition 3.5. The class of *partial recursive functions* is the smallest class of functions for which the following hold.

(1) This class contains all the initial functions.
(2) This class is closed under composition, primitive recursion and minimization.

Definition 3.6. The class of *total recursive functions* is the class of all partial recursive functions that are total.

Remark 3.0.1. We can think of the class of primitive recursive functions as the class of functions described by only for loops, and we can think of partial recursive functions as those functions which are described by both for loops and while loops.

Example 3.1. Let S be the successor function. Observe that the function

$$f(n) = n + 2$$

can be written as $f = S \circ S$, and hence f is a primitive recursive function.

Example 3.2. Consider the function $plus : \mathbb{N}^2 \to \mathbb{N}$ given by

$$plus(n, m) = n + m$$

We claim that $plus$ is a primitive recursive function. Observe that

$$\begin{aligned} plus(0, m) &= g(m) = \pi_1^1(m) \\ plus(n + 1, m) &= h(n, plus(n, m), m) = S \circ \pi_2^3(n, plus(n, m), m) = (n + 1) + m \end{aligned}$$

Example 3.3. Consider the function $mult : \mathbb{N}^2 \to \mathbb{N}$ given by

$$mult(n, m) = mn$$

We claim that $mult$ is a primitive recursive function. Observe that

$$\begin{aligned} mult(0, m) &= Z(m) = 0 \\ mult(n + 1, m) &= (plus \circ (\pi_2^3, \pi_3^3))(n, mult(n, m), m) = nm + m + (n + 1)m \end{aligned}$$

Example 3.4. Consider the function exponential function

$$exp(n, m) = m^n$$

We have

$$\begin{aligned} exp(0, m) &= (S \circ Z)(m) = 1 \\ exp(n + 1, m) &= (mult \circ (\pi_2^3, \pi_3^3))(n, mult(n, m), m) = m^n \cdot m = m^{n+1} \end{aligned}$$

Example 3.5. Let us try to define the predecessor function, i.e

$$\begin{aligned} pred(0) &= 0 \\ pred(n + 1) &= n \end{aligned}$$

First, consider the auxilliary function $pred'$ defined by

$$\begin{aligned} pred'(0, m) &= 0 \\ pred(n + 1, m) &= n \end{aligned}$$

i.e $pred'$ ignores the second argument, and on the first argument it is the predecessor function. Observe that

$$\begin{aligned} pred'(0, m) &= Z(m) = 0 \\ pred'(n + 1, m) &= \pi_1^3(n, pred'(n, m), m) = n \end{aligned}$$

which implies that $pred'$ is a primitive recursive function. Now, we can write $pred$ as a composition

$$pred(n) = pred'(n, Z(n))$$

and hence $pred$ is a primitive recursive function.

Example 3.6. Let us define the function $subtr(m, n) = m - n$. More specifically,

$$subtr(m, n) = \begin{cases} 0 & , \quad m \leq n \\ m - n & , \quad \text{otherwise} \end{cases}$$

Consider the auxilliary function $subtr'$ given by $subtr'(n,m) = m-n$. Now observe that

$$\begin{aligned} subtr'(0,m) &= m \\ subtr'(n+1,m) &= pred(subtr'(n,m)) = (pred \circ \pi^2_3)(n, subtr'(n,m), m) \end{aligned}$$

which means that $subtr'$ is a primitive recursive function. Now, observe that

$$subtr = subtr' \circ (\pi^2_2, \pi^2_1)$$

and hence $subtr$ is also a primitive recursive function.

Example 3.7. The function $f(m) = \log_2(m)$ is defined by minimization from $g(n,m) = m - 2^n$ (note that we have already shown that the exponential and subtraction functions are primitive recursive).

3.2. **Encoding Recursive Functions.**

In this section, we will try and encode recursive functions.

3.2.1. *The Zero Function.*

Observe that

$$[Z] = \lambda x.[0]$$

and it can be easily seen that $[Z][n] \xrightarrow{*} [0]$ for any $n \in \mathbb{N}$.

3.2.2. *The Successor Function.*

We already have an encoding for this:

$$[succ] = \lambda pfx.f(pfx)$$

3.2.3. *The Projection Function.*

The projection function is encoded as

$$[\pi^k_i] = \lambda x_1 x_2 \cdots x_k.x_i$$

3.2.4. *Function Composition.*

If $f : \mathbb{N}^k \to \mathbb{N}$ is defined by $f = g \circ (h_1, ..., h_l)$ then

$$[f] = \lambda x_1 x_2 \cdots x_k.[g]([h_1]x_1x_2\cdots x_k)\cdots([h_l]x_1x_2\cdots x_k)$$

3.2.5. *Primitive Recursion.*

Suppose f is defined via primitive recursion from g and h. To encode primitive recursion into a λ-expression, we need to first get rid of recursion, and convert it into iteration. Observe that by definition,

$$\begin{aligned} f(0,\vec{x}) &= g(\vec{x}) \\ f(i+1,\vec{x}) &= h(i, f(i,\vec{x}), \vec{x}) \end{aligned}$$

Given l and $\vec{n}$, the idea will be to generate a sequence of pairs

$$(0,a_0),(1,a_1),...,(l,a_l)$$

where $a_0 = g(\vec{n})$ and $a_{i+1} = h(i, a_i, n)$. So let $t(i) = (i, a_i)$. So we have

$$t(0) = (0,a_0) = (0, g(\vec{n}))$$

and that

$$\begin{aligned} t(i+1) = (i+1, a_{i+1}) &= (i+1, h(i, a_i, \vec{n})) \\ &= (succ(fst(t(i))), h(fst(t(i)), snd(t(i)), \vec{n})) \end{aligned}$$

(where fst returns the first coordinate of a pair, and similarly snd returns the second coordinate of a pair) and clearly $f(l,\vec{n})$ can be retrieved as $snd(t(l))$. The key point is that we can generate the $t(i)'$s by *iteration*. To do this, define $Init = (0, g(\vec{n}))$ and

$$Step(t(i)) = t(i+1)$$

which implies that $t(l) = Step^l(Init)$ and hence

$$f(l, \vec{n}) = snd(t(l)) = snd(Step^l(Init))$$

To do this, first let us encode a pair. This is simply done as follows:

$$[pair] = \lambda xyz.zxy$$

With this encoding, we see that

$$[pair]\ a\ b \xrightarrow{*} \lambda z.zab$$

We now encode $[fst]$ and $[snd]$ as follows.

$$[fst] = \lambda p.p(\lambda xy.x)$$
$$[snd] = \lambda p.p(\lambda xy.y)$$

Then observe that

$$\begin{aligned}[fst]\ ([pair]\ a\ b) &\xrightarrow{*} [fst]\ (\lambda z.zab)\\ &\xrightarrow{*} (\lambda p.p(\lambda xy.x))(\lambda z.zab)\\ &\xrightarrow{*} (\lambda z.zab)(\lambda xy.x)\\ &\xrightarrow{*} (\lambda xy.x)\ a\ b\\ &\xrightarrow{*} (\lambda y.a)b\\ &\xrightarrow{*} a\end{aligned}$$

Similarly, one can show that

$$[snd]\ ([pair]\ a\ b) \xrightarrow{*} b$$

Now, we can encode $Init$ as

$$[Init] = [pair][0]([g]x_1 \cdots x_k)$$

(here $x_1 \cdots x_k$ serves as the vector $\vec{n}$). The $Step$ function can be encoded as

$$[Step] = \lambda y.[pair]([succ]([fst]\ y))([h]\ ([fst]\ y)\ ([snd]\ y)x_1 \cdots x_k)$$

Finally, put

$$[f] = \lambda xx_1 \cdots x_k.[snd](x[Step][Init])$$

Above, x denotes the number of times to iterate, and $x_1 \cdots x_k$ is the input vector $\vec{n}$. So, we have that

$$[f][l][n_1] \cdots [n_k] \xrightarrow{*} [f(l, \vec{n})]$$

Now, we can encode the scheme of primitive recursion using λ-calculus as follows.

$$\begin{aligned}[PR] =& \lambda hgxx_1 \cdots x_k.[snd](x(\lambda y.[pair]([succ]([fst]\ y))([h]\ ([fst]\ y)\ ([snd]\ y)x_1 \cdots x_k))\\ &([pair][0]([g]x_1 \cdots x_k)))\end{aligned}$$

So, $[PR]\ [h]\ [g]$ will return the encoding of f, where f is defined via primitive recursion from h and g.

3.2.6. *Booleans.* The encodings for booleans in λ-calculus are as follows.

$$[true] = \lambda xy.x$$
$$[false] = \lambda xy.y$$

3.2.7. *If-Then-Else.* Consider the following encoding.

$$[if-then-else] = \lambda bxy.bxy$$

Then observe that

$$\begin{aligned} [if-then-else]\ [true]\ M\ N &\xrightarrow{*} (\lambda xy.(\lambda pq.p)xy)\ M\ N \\ &\xrightarrow{*} (\lambda xy.(\lambda q.x)y)\ M\ N \\ &\xrightarrow{*} (\lambda xy.x)\ M\ N \\ &\xrightarrow{*} M \end{aligned}$$

Similarly, one can show that

$$[if-then-else]\ [false]\ M\ N \xrightarrow{*} N$$

As a matter of syntactic sugaring, we write $if\ b\ then\ f\ else\ g$ in place of $[if - then - else]\ b\ f\ g$.

3.3. **Encoding a test for zero.** Using our encoding of booleans, we can encode a test for checking whether the input is zero or not. Consider the following:

$$[iszero] = \lambda x.x(\lambda z.[false])[true]$$

Then, we see that

$$[iszero]\,[n] \xrightarrow{*} [n](\lambda z.[false])[true] \xrightarrow{*} (\lambda z.[false])^n[true]$$

So, if n is zero, $[iszero]\,[n]$ reduces to $[true]$, and otherwise it reduces to $[false]$.

3.4. **Encoding μ-recursion.** Let $g : \mathbb{N} \times \mathbb{N}^k \to \mathbb{N}$ be some function. Recall that f is defined from g via μ-recursion if

$$f(\vec{n}) = \mu n.(g(n, \vec{n}) = 0)$$

First, define

$$W = \lambda y.\ if\ ([iszero]([g]yx_1 \cdots x_k))\ then\ (\lambda w.y)\ else\ (\lambda w.w([succ]\,y)\,w)$$

Using W, we claim that the encoding for f is

$$[f] = \lambda x_1 \cdots x_n.W\,[0]\,W$$

Observe that

$$[f][n_1] \cdots [n_k] \xrightarrow{*}_\beta W'\,[0]\,W'$$

where $W' = W[x_1 := [n_1], ..., x_k := [n_k]]$. Now suppose $g(i, \vec{n}) = 0$. Then we see that

$$[g][i][n_1] \cdots [n_k] \xrightarrow{*} [0]$$

and hence

$$[iszero]([g][i][n_1] \cdots [n_k]) \xrightarrow{*} [iszero][0] \xrightarrow{*} [true]$$

So,

$$\begin{aligned} W'[i]W' &\xrightarrow{*} (if\ ([iszero]([g][i][n_1] \cdots [n_k]))\ then\ (\lambda w.[i])\ else\ (\lambda w.w([succ]\,[i])\,w))\,W' \\ &\xrightarrow{*} (if\ [true]\ then\ (\lambda w.[i])\ else\ (\lambda w.w([succ]\,[i])\,w))\,W' \\ &\xrightarrow{*} (\lambda w.[i])\,W' \\ &\xrightarrow{*} [i] \end{aligned}$$

On the other hand, if $g(i, \vec{n}) = m > 0$, then one can similarly show that

$$W'\,[i]\,W' \xrightarrow{*} (\lambda w.w([succ][i])w)\,W' \xrightarrow{*} W'\,[i+1]\,W'$$

So, if $g(b, \vec{n}) = 0$ and $g(a, \vec{n}) > 0$ for all $a < b$ then

$$W'\,[0]\,W' \xrightarrow{*} W'\,[1] \xrightarrow{*} \cdots \xrightarrow{*} W'\,[b]\,W' \xrightarrow{*} [b]$$

and hence

$$[f][n_1]\cdots[n_k] \xrightarrow{*} [b]$$

which shows that the given encoding indeed is the correct encoding for f.

In general, the expression

$$[Mu] = \lambda g x_1 \cdots x_k.U\,[0]\,U$$

encodes the schema of μ-recursion, where

$$U = W = \lambda y.\ if\ ([iszero]([g]yx_1\cdots x_k))\ then\ (\lambda w.y)\ else\ (\lambda w.w([succ]\,y)\,w)$$

3.5. **Normal Terms.** A term Q is said to be *normal* if there is no R such that $Q \xrightarrow{*} R$. Another way of saying this is

$$\forall R:\ Q \xrightarrow{*} R \implies Q = R$$

Q is said to be a normal form of M if $M \xrightarrow{*} Q$ and Q is a normal term.

Proposition 3.1 (**Church-Rosser Property**). *If $M \xrightarrow{*} N$ and $M \xrightarrow{*} P$ then there exists Q such that $N \xrightarrow{*} Q$ and $P \xrightarrow{*} Q$.*

Proof. To be completed. ■

Corollary 3.1.1. *Normal forms are unique.*

3.6. **Recursive definitions and the Y combinator.** Define

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Then we see that

$$Y\,F \xrightarrow{*} (\lambda x.F(xx))(\lambda x.F(xx)) \xrightarrow{*} F((\lambda x.F(xx))(\lambda x.F(xx)))$$

Also, using only the first half of the reductions above, we see that

$$F(Y\,F) \xrightarrow{*} F((\lambda x.F(xx))(\lambda x.F(xx)))$$

In other words, there is some G such that $Y\,F \xrightarrow{*} G$ and $F(Y\,F) \xrightarrow{*} G$. In this case, we say that $Y\,F =_\beta F(Y\,F)$. So, we have shown that for any F, $Y\,F$ is a C such that $C =_\beta F\,C$.

3.7. **The Θ combinator.** We now consider the following problem: given a λ-expression F, we need to find an expression C such that

$$C \xrightarrow{*} F\,C$$

Define

$$\Theta = (\lambda xy.y(xxy))(\lambda xy.y(xxy))$$

We then observe that

$$\begin{aligned}\Theta\, F &= (\lambda xy.y(xxy))(\lambda xy.y(xxy))\, F \\ &\xrightarrow{*} (\lambda y.y((\lambda xy.y(xxy))(\lambda xy.y(xxy))y))\, F \\ &\xrightarrow{*} F((\lambda xy.y(xxy))(\lambda xy.y(xxy))F) \\ &= F(\Theta\, F)\end{aligned}$$

and in short, we can write

$$\Theta\, F \xrightarrow{*} F(\Theta\, F)$$

Remark 3.1.1. The Y and Θ combinators are also called *fixed-point combinators*.

3.8. η **rule.** Observe that if x does not occur free in M, then for all N,

$$(\lambda x.(Mx))N \to MN$$

So, $\lambda x.(Mx)$ behaves just like M. So, we have a new reduction rule, called η-*reduction*: if $x \notin FV(M)$, then

$$\lambda x.(Mx) \underset{\eta}{\to} M$$

4. Recursive functions and Turing Computability

4.1. **Exactly what functions are recursive.** In the previous sections, we have shown that for every recursive function $f : \mathbb{N}^k \to \mathbb{N}$ there is a λ-calculus expression $[f]$ such that

$$[f][n_1]\cdots[n_k] \xrightarrow{*} [f(n_1, ..., n_k)]$$

for all $n_1, ..., n_k \in \mathbb{N}$. Further, if $[f][n_1]\cdots[n_k] \xrightarrow{*} [m]$ for any $m \in \mathbb{N}$, then

$$m = f(n_1, ..., n_k)$$

and this is an easy consequence of the **Church-Rosser** 3.1 property and the fact that $[n]$ is in normal form, for any $n \in \mathbb{N}$ (which is not hard to show). Now, we will explore exactly what functions are recursive.

4.2. **Writing programs for recursive functions.** In this section, we will informally show that Recursive functions Turing-computable (i.e, computable by Turing Machines) by showing that we can write programs for recursive functions.

(1) It is trivial to write programs for initial functions: the zero function, the successor function and the projection function.
(2) Composition: if $f : \mathbb{N}^k \to \mathbb{N}$ is defined by $f = g \circ (h_1, ..., h_l)$, then the following program works for f.

```
function f(x1, x2, ..., xk){
   y1 = h1(x1, x2, ..., xk);
   y2 = h2(x1, x2, ..., xk);
   ...
   yl = hl(x1, x2, ..., xk);
   return g(y1, y2, ..., yl);
}
```

(3) Primitive recursion: If $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is defined from $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ by primitive recursion, then the following program works for f.

```
result = g(n1, ..., nk) //f(0, n1, ..., nk)
for (i = 0; i < n; i++){ //computing f(i + 1, n1, ..., nk)
    result = h(i, result, n1, ..., nk);
}
return result;
```

(4) μ-recursion: if $f : \mathbb{N}^k \to \mathbb{N}$ is defined from $g : \mathbb{N}^{k+1} \to \mathbb{N}$ by μ-recursion, then we have the following.

```
i = 0;
while (g(i, n1, ..., nk) > 0) {i = i + 1;}
return i;
```

So, it follows that a program can be written for any recursive function.

4.3. **Primitive Recursive Relations.** A relation $R \subseteq \mathbb{N}^k$ is said to be *primitive recursive* if its characteristic function $c_R : \mathbb{N}^k \to \{0, 1\}$ is primitive recursive.

Example 4.1. The relation $iszero \subseteq \mathbb{N}$ is primitive recursive, since its characteristic function c_{iszero} is primitive recursive:

$$c_{iszero}(0) = succ(\pi_1^1(0))$$
$$c_{iszero}(n + 1) = Z(n)$$

Example 4.2. The relation $x \leq y$ on natural numbers is also primitive.