λn.n(λx. false)(true)

Subtraction and Comparison

- Subtraction: m-n
  - λm. λn. nPRED m
- Comparison
  - λn. λm. isZero(subtract n m)
  - If the predecessor function applied n times to m yields zero, then it is true that??
    - *greaterOrEqual*
  - *lessOrEqual*= λn. λm. isZero(subtract m n)

Division

```
if(a>=b) then
     return 1+ (a-b)/b);
else
   return 0
```

if_then_else$=_{def}$ *λcond.λthen_do*. *λelse_do*. Cond (*then_do*) (*else_do*)

The problem here is that we need to express recursion without explicitly calling itself

- *a/b*
  - *if a>=b then 1+ (a-b)/b else 0*
- divide=λa. λb. if_then_else(*greaterOrEqual a b*) (suc *b)* (zero)
- divide= λa. λb. if_then_else(*greater b a)* (zero) (succ *(self (subtract a b) b)*
  - divide seven three
  - if_then_else(*greaterOrEqual seven three*) (succ(*self (subtract seven three) three*) (zero)
  - (succ(*self (subtract seven three) three)*
  - *(succ (*if_then_else(*greaterOrEqual four three*) (succ(*self (subtract four three) three*) (zero)))

Little bit of creativity + little bit of elegance

- Self application
  - sa = $\lambda x.\ x\ x$
- This function takes an argument x, which is apparently a function

- Loop : $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

- $\Omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$
- The Omega Combinator is just the simplest function which infinitely recurs without calling itself.

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

https://youtu.be/BC8ZAMwfwi4

Y Combinator

$Yt=t(Yt)=t(t(Yt))=\ldots$

- Y combinator can be defined as
  - $Y = \lambda t.\ (\lambda x.\ t\ (x\ x))\ (\lambda x.\ t\ (x\ x))$
- $Yz=(\lambda t.\ (\lambda x.\ t\ (x\ x))\ (\lambda x.\ t\ (x\ x)))z$
- $=(\lambda x.\ z\ (x\ x))\ (\lambda x.\ z\ (x\ x))$

$Yz =(\lambda x.\ z\ (x\ x))\ (\lambda g.\ z\ (g\ g))$

$= z\ (\lambda g.\ z\ (g\ g))\ (\lambda g.\ z\ (g\ g))$

$=z(Yz)$

$=z((\lambda g.\ z\ (g\ g))\ (\lambda h.\ z\ (h\ h)))$

$=z(z((\lambda h.\ z\ (h\ h))\ (\lambda h.\ z\ (h\ h))))$

$=z(z(Yz)\ldots$

Y Combinator

- Y combinator can be defined as
  - Y = λf. (λ x. f(x x)) (λx. f (x x))
- Yf=f(Yf)=f(f(Yf))=…

- Fact:=Y (λ f. λ n. `If_then_else` (isZero n) one (mult n(f(pred n))

Calculating factorial

- T= `(λ f. λ n. If_then_else (isZero n) one (mult n(f(pred n))`
- **`Fact=YT  Fact 2= (YT) two`**
- **`=T(YT)two`**
- **`= (λ f. λ n. If_then_else (isZero n) one (mult n(f(pred n)) (YT) two`**
- …

`divide=λa. λb. if_then_else(greaterOrEqual a b) (succ (self (subtract a b) b) (zero)`

Division again!

- **`D:= λf. λa. λb. if_then_else(greaterOrEqual b a) (Zero)(succ(f (subtract a b) b)`**
- **`YD=D(YD)`**
- **`YD five two`**
- **`=> D(YD) five two`**
- **`=> if_then_else(greaterOrEqual two five) (Zero) (succ(YD (subtract five two) two)`**
- **`=> succ(YD three two)`**
- …

Summation

To compute sum of natural numbers from 0 to n $\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i.$

$$\mathsf{R} \equiv (\lambda rn.\mathsf{Z}n0(n\mathsf{S}(r(\mathsf{P}n))))$$

Fibonacci Series

- F(n)=f(n-1)+f(n-2) if n>2
-         =1   else

- Y := λf.(λx.f (x x)) (λx.f (x x))
- F := Y (λf. λn.if_then_else (LessThanOrEqual two) one (f(Pred n) succ f(Pred(pred n))))

Tail Recursion

Tail recursion is a situation where a recursive call is the last thing a function does before returning, and the function either returns the result of the recursive call or (for a procedure) returns no result. A compiler can recognize tail recursion and replace it by a more efficient implementation.

Recursive sum

```
function recsum(x) {
    if (x === 0) {
        return 0;
    } else {
        return x + recsum(x - 1);
    }
}
```

```
recsum(5)
5 + recsum(4)
5 + (4 + recsum(3))
5 + (4 + (3 + recsum(2)))
5 + (4 + (3 + (2 + recsum(1))))
5 + (4 + (3 + (2 + (1 + recsum(0)))))
5 + (4 + (3 + (2 + (1 + 0))))
5 + (4 + (3 + (2 + 1)))
5 + (4 + (3 + 3))
5 + (4 + 6)
5 + 10
15
```

```
tailrecsum(5, 0)
tailrecsum(4, 5)
tailrecsum(3, 9)
tailrecsum(2, 12)
tailrecsum(1, 14)
tailrecsum(0, 15)
15
```

```
function tailrecsum(x, running_total = 0) {
    if (x === 0) {
        return running_total;
    } else {
        return tailrecsum(x - 1, running_total + x);
    }
}
```

- If the continuation is empty and there are no backtrack points, nothing need be placed on the stack; execution can simply jump to the called procedure, without storing any record of how to come back. This is called LAST–CALL OPTIMIZATION

- A procedure that calls itself with an empty continuation and no backtrack points is described as TAIL RECURSIVE, and last–call optimization is sometimes called TAIL–RECURSION OPTIMIZATION

factorial

```
Fact(acc,n) {
   return n==1?acc:Fact(acc*n,n-1);
}
```

T= (λ f. λ n. If_then_else (isZero n) one (mult n(f(pred n))

Introduction to Typed Lambdas

- A type is a collection of computational entities that share some common property.
- A type specifies a class of objects and associated operations
- For example, the type int represents all expressions that evaluate to an integer
- The type int → int represents all functions from integers to integers.
- The difference between types and sets is that types are *syntactic* objects, i.e., we can speak of types without having to speak of their elements. We can think of types as *names* for sets

Introducing types

- Even though the lambda calculus is untyped, a large majority of the lambda terms that we look at can be given types
- In fact, looking at the types of the terms provides insight into the kind of functions these terms represent
- So, wherever possible, we mention the types of the functions. We use capital letters A, B, . . . to represent arbitrary types and the → symbol to represent function types.
- A → B represents the type of functions from A to B, i.e., functions that given A-typed arguments, return B-typed results.
- We use a bracketing convention to parse type expressions with multiple → symbols

Simple types: $A, B ::= \iota \mid A \to B \mid A \times B \mid 1$

Type definition

- Greek letter ι (“iota”) to denote a basic type
- The base types are things like the type of integers or the type of Booleans
- The type $A \to B$ is the type of functions from A to B.
- The type $A \times B$ is the type of pairs <x, y>, where x has type A and y has type B
- The type 1 is a one-element type.
  - You can think of 1 as an abridged version of the booleans, in which there is only one boolean instead of two.
  - You can think of 1 as the “void” or “unit” type in many programming languages: the result type of a function that has no real result.

- When we write types, we adopt the convention that $\times$ binds stronger than $\to$, and $\to$ associates to the right.
- $A \times B \to C$
- is $(A \times B) \to C$
- $A \to B \to C$
- is $A \to (B \to C)$

Introducing Types

- We are going to construct functions to represent typed objects
- In general, an object will have a type and a value
- We need to be able to:
  - i) construct an object from a value and a type
  - ii) select the value and type from an object
  - iii) test the type of an object

- We will represent an object as a type/value pair
- *def make_obj type value = λs.(s type value)*

https://sookocheff.com/post/fp/typed-lambda-calculus/
Chapter 5, AN INTRODUCTION TO FUNCTIONAL ROGRAMMING THROUGH LAMBDA CALCULUS
Greg Michaelson, Heriot-Watt University, Edinburgh, UK

Extracting type and/or value

```
def selectSecond=λfirst.λsecond.second
def value obj =obj selectSecond
```

- `def selectFirst=λfirst. λsecond. first`
- `def type obj=obj selectFirst`
- we can use these functions to define a (type, value) pair and then access the type
- `def myObj ⟨type⟩ ⟨value⟩=λs.(s ⟨type⟩⟨value⟩)`

- `type myObj=myObj selectFirst`
- `         =λs.(s ⟨type⟩⟨value⟩) selectFirst`
- `         =(selectFirst ⟨type⟩⟨value⟩)`
- `        =(λfirst.λsecond.first ⟨type⟩ ⟨value⟩)`
- `        =(λsecond.⟨type⟩) ⟨value⟩`
- `        =⟨type⟩`
- Once types are defined, however, we should only manipulate typed objects with typed operations to ensure that the type checks aren't overridden.

Type Boolean

- We will represent the boolean type as one:
- def bool_type = one
- Constructing a boolean type involves preceding a boolean value with bool_type:
- def MAKE_BOOLEAN = make_obj bool_type
- which expands as:
- λvalue. λs.(s bool_type value)
- We can now construct the typed booleans TRUE and FALSE from the untyped versions by:
- def TRUE = MAKE_BOOLEAN true
- which expands as:
- λs.(s bool_type true)

Type boolean

- def FALSE = MAKE_BOOLEAN false
- which expands as:
- λs.(s bool_type false)
- The test for a boolean type involves checking for bool_type:
- def isbool = istype bool_type
- This definition expands as:
- λobj.(equal (type obj) bool_type)

We will use numbers to represent types and numeric comparison to test the type

```
def istype t obj = equal (type obj) t
```

Self application in typed lambda calculus

- Even though self-application allows calculations using the laws of the lambda calculus, what it means conceptually is not at all clear
- We can see some of the problems by just trying to give a type to sa = $\lambda x.\ x\ x$.
- Suppose the argument x is of type A.
- But, since x is being applied as a function to x, the type of x should be of the form $A \rightarrow \ldots$.
- How can x be of type A as well as $A \rightarrow B \ldots$?
- Is there a type A such that $A = (A \rightarrow B)$?
- In traditional mathematics (set theory), there is no such type.
- The concept of "domains" which can be used to represent types (instead of traditional sets)
- This led to the development of an elegant theory of domains, which serves as the foundation for the mathematical meaning of programming languages.

https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/lambda-calculus-handout.pdf

Objects in lambda calculus

- Self application is used very fundamentally in implementing object-oriented programming languages. Suppose we have an object x with a method m.
- We might invoke this method by writing something like x.m(y).
- Inside the method m, there would be references to keywords like “self” or “this” which are supposed to represent the object x itself.
- One way of solving the problem is to translate the method m into a function m’ that takes two arguments: in addition to the proper argument y, the object on which the method is being invoked. So, the definition of m’ looks like:
- m' = λ self. λ y. . . . the body of m . . .
- The method call x.m(y) is then translated as x.m’ (x)(y).

Objects in lambda calculus

- The object x has a collection of such functions encoding the methods.
- The method call x.m(y) is then translated as x.m’(x)(y).
- This is a form of self application.
- The function m’, which is a part of the structure x, is applied to the structure x itself.

Expressiveness of Lambda Calculus

- The λ-calculus can express
  - data types (integers, booleans, lists, trees, etc.)
  - branching (using booleans)
  - recursion
- This is enough to encode Turing machines
- Encodings can be done
- But programming in pure λ-calculus is painful
  - add constants (0, 1, 2, …, true, false, if-then-else, etc.)
  - add types