Lambda Calculus and LISP

Lambda Calculus: First Functional Language

- Church, A., 1932, "A set of postulates for the foundation of logic", *Annals of Mathematics* (2nd Series), 33(2): 346–366.

Example

Scala equivalent	**Lambda calculus**
(((x : Any) => (y : Any) => x)(a))(b)	(λxy. x) a b

Lambda calculus has only variables (x,y,a,b,…) and these two constructs:

	Scala equivalent	**Lambda calculus**
application	f(x)	f x
abstraction	(x:Any) => M	λx.M

The main rule: argument substitution (β-reduction)

Functions have one argument. We use abbreviations such as these:

$\lambda xy.MN$	=	$\lambda x.(\lambda y.(MN))$	similar to (x,y) => M(N)
f M N	=	((f M) N)	similar to f(M,N)

We do not worry about types in the (untyped) λ calculus

Example of applying β-reduction (special case of Lecture 1 substitution model):

$(\lambda x.M)N \Rightarrow_\beta$ "term obtained from M by replacing all x occurrences with N"

- $(\lambda x.\ x)\ (a\ b) \Rightarrow_\beta (a\ b)$
- $(\lambda xy.\ c\ x)\ a\ b = ((\lambda x.\ (\lambda y.\ (c\ x)))\ a)\ b \Rightarrow_\beta (\lambda y.\ (c\ a))\ b \Rightarrow_\beta c\ a$

- $(\lambda\ f\ x.\ f(f\ x))\ (\lambda y.\ a)\ b \Rightarrow_\beta (\lambda y.\ a)((\lambda y.\ a)\ b) \Rightarrow_\beta a$

λ calculus can do: Booleans

Given hypothetical if statement   if (b) M N   represent Boolean values as the functions corresponding to "if (b)" code fragment

if (true) M N	should be	M
if (false) M N	should be	N

Define

true $= \lambda x\, y.\, x$ true M N $= (\lambda x\, y.\, x)$ M N $\Rightarrow_\beta$ M

false $= \lambda x\, y.\, y$ false M N $= (\lambda x y.\, y)$ M N $\Rightarrow_\beta$ N

So instead of "if (b) M N" we just write (b M N)

λ calculus can do: Pairs

Pair is something from which we can get the first and the second element

Define

$$(M,N) = \lambda f.\ f\ M\ N$$

$$P.\_1 = P\ (\lambda x\ y.\ x)$$

$$P.\_2 = P\ (\lambda x\ y.\ y)$$

Why does this work?

$$(M,N).\_1 = (\lambda f.\ f\ M\ N)\ (\lambda x\ y.\ x) \Rightarrow_\beta (\lambda x\ y.\ x) M\ N \Rightarrow_\beta M$$

$$(M,N).\_2 = \underline{(\lambda f.\ f M N)}\ (\lambda x y.\ y) \Rightarrow_\beta (\lambda x y.\ \underline{y}) M N \Rightarrow_\beta N$$

λ calculus can do: Lists

A list is something we can match on and deconstruct if it is not empty:

```
list match {
  case Nil => M
  case Cons(x,y) => N(x,y)
}
```

A list value is given by how it interacts with two terms **M** and **N**

We define it as a function that will take such **M** and **N** as arguments

Nil = λmn. m   Nil M N $\Rightarrow_\beta$ M

Cons(P,Q) = λmn. n (P,Q)

Cons(P,Q) M (λp. p._1) $\Rightarrow_\beta$ (λmn. n (P,Q)) M (λp. p._1) $\Rightarrow_\beta$ (λp. p._1) (P,Q) $\Rightarrow_\beta$ (P,Q)._1

Cons is like a pair, but takes m as argument, too, to fit along with Nil

Returning pair (tail,tail) if list non-empty, else Z

```
list match {
  case Nil => Z
  case Cons(x,y) => (y,y)
}
```

Becomes nothing else but

$$\text{list } Z\ (\lambda p.\ (\lambda y.\ (y,y))\ \ (p.\_2))$$

i.e.

$$\text{list } Z\ (\lambda p.\ (\lambda y.\ \lambda f.\ f\ y\ y)\ \ (p\ (\lambda u\ v.\ v)))$$

Computation that takes any number of steps

$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \Rightarrow_\beta (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \Rightarrow_\beta \ldots$ loops.

More usefully: $(\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x)) \Rightarrow_\beta F\ ((\lambda x.\ F\ (x\ x))(\lambda x.\ F\ (x\ x)))$

If we denote $Y_F = (\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x))$ **(for each term F)**

Then $Y_F \Rightarrow_\beta F\ ((\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x))) = F\ Y_F$ i.e. $Y_F \Rightarrow_\beta F(Y_F)$

A recursive function uses itself in its body (typically applies it):

def h(x:Any) = P(h(Q(x)),x) for some P and Q

def h = ((x:Any) => P(h(Q(x)),x))

Denote right-hand side of the last **def** by F(h), since x is a bound variable

def h = F(h) to unfold recursion, replace h by F(h) in body

We define $h = Y_F$ so $h = Y_F \Rightarrow_\beta F\ Y_F \Rightarrow_\beta F(F\ Y_F) = F(F\ h) \Rightarrow_\beta \ldots$

Replace all list elements by Z: List(1,2,3) → List(Z,Z,Z)

```
def mkZ(list) = list match {
  case Nil => Nil
  case Cons(x,y) => Cons(Z, mkZ(y))
}
```

After encoding match, still using recursion

$$\text{mkZ} = \lambda\text{list. list Nil } (\lambda\text{ p. Cons(Z, mkZ(p.\_2)))}$$

After encoding recursion, it becomes mkZ = Y_F

for $\quad F = \lambda\text{self. } \lambda\text{list. list Nil } (\lambda\text{ p. Cons(Z, self(p.\_2)))}$

So mkZ can be defined as Y_F which in this case is:

$$(\lambda x.\ (\lambda\text{ self. } \lambda\text{list. list Nil } (\lambda\text{ p. Cons(Z, self(p.\_2))))}\ \ (x\ x))$$
$$\ (\lambda x.\ (\lambda\text{ self. } \lambda\text{list. list Nil } (\lambda\text{ p. Cons(Z, self(p.\_2))))}\ \ (x\ x))$$