Lambda Calculus and LISP
Lambda Calculus: First Functional Language


Example

<table>
<thead>
<tr>
<th>Scala equivalent</th>
<th>Lambda calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>(((x : Any) =&gt; (y : Any) =&gt; x)(a))(b)</td>
<td>($\lambda xy. x$) a b</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Scala equivalent</th>
<th>Lambda calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>application</td>
<td>f(x)</td>
</tr>
<tr>
<td>abstraction</td>
<td>(x:Any) =&gt; M</td>
</tr>
<tr>
<td></td>
<td>$\lambda x. M$</td>
</tr>
</tbody>
</table>
The main rule: argument substitution ($\beta$-reduction)

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

$$\lambda xy. MN = \lambda x.(\lambda y. (MN))$$  
$$f M N = ((f M) N)$$

similar to  

$$\text{(x,y) } \Rightarrow M(N)$$  

$$\text{f(M,N)}$$

We do not worry about types in the (untyped) $\lambda$ calculus

Example of applying $\beta$-reduction (special case of Lecture 1 substitution model):

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

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

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

- $$(\lambda f \ x. f(f \ x)) \ (\lambda y. \ a) \ b \Rightarrow_\beta (\lambda y. \ a)((\lambda y. \ a) \ b) \Rightarrow_\beta a$$
λ calculus can do: Booleans

Given hypothetical if statement \( \text{if} (\text{b}) \, M \, N \) represent Boolean values as the functions corresponding to “if (b)” code fragment

\[
\begin{align*}
\text{if (true)} \, M \, N & \quad \text{should be} \quad M \\
\text{if (false)} \, M \, N & \quad \text{should be} \quad N
\end{align*}
\]

Define

\[
\begin{align*}
\text{true} & \quad = \lambda x \, y. \, x \\
\text{false} & \quad = \lambda x \, y. \, y
\end{align*}
\]

\[
\begin{align*}
\text{true} \, M \, N & \quad = \quad (\lambda x \, y. \, x) \, M \, N \quad \Rightarrow_\beta \quad M \\
\text{false} \, M \, N & \quad = \quad (\lambda x \, y. \, y) \, M \, N \quad \Rightarrow_\beta \quad N
\end{align*}
\]

So instead of “if (b) M N” we just write \( (b \, M \, N) \)
\( \lambda \) 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?

\[
(M,N)._1 = (\lambda f. f M N) (\lambda x y. x) \Rightarrow_{\beta} (\lambda x y. x)M N \Rightarrow_{\beta} M
\]

\[
(M,N)._2 = (\lambda f. f MN) (\lambda x y. y) \Rightarrow_{\delta} (\lambda x y. y) MN \Rightarrow_{\delta} N
\]
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

- $\text{Nil} = \lambda mn. \text{m}$
- $\text{Cons}(P,Q) = \lambda mn. \text{n } (P,Q)$

$\text{Cons}(P,Q) M (\lambda p. p._1) \Rightarrow_B (\lambda mn. \text{n } (P,Q)) M (\lambda p. p._1) \Rightarrow_B (\lambda p. p._1) (P,Q) \Rightarrow_B (P,Q)._1$

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
```
list Z (\p. (\y. (y,y)) (p._2))
```

i.e.
```
list Z (\p. (\y. \f. f y y) (p (\u v. v)))
```
Computation that takes any number of steps

\[(\lambda x. x x) (\lambda x. x x) \Rightarrow_\beta (\lambda x. x x) (\lambda x. x x) \Rightarrow_\beta \ldots \text{ loops.}\]

More usefully:

\[(\lambda x. \text{F}(x x)) (\lambda x. \text{F}(x x)) \Rightarrow_\beta \text{F} (((\lambda x. \text{F}(x x))(\lambda x. \text{F}(x x)))\]

If we denote \(Y_F = (\lambda x. \text{F}(x x)) (\lambda x. \text{F}(x x))\) (for each term \(F\))

Then \(Y_F \Rightarrow_\beta \text{F} (((\lambda x. \text{F}(x x))(\lambda x. \text{F}(x x))) = \text{F} Y_F\) i.e. \(Y_F \Rightarrow_\beta \text{F}(Y_F)\)

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

```python
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 \(\text{def}\) by \(F(h)\), since \(x\) is a bound variable

```python
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:  \( \text{List}(1,2,3) \rightarrow \text{List}(Z,Z,Z) \)

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

After encoding match, still using recursion

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

After encoding recursion, it becomes \( \text{mkZ} = Y_F \)

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

So \( \text{mkZ} \) can be defined as \( Y_F \) which in this case is:

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