We now look at lambda calculus, the theoretical stuff that underlies functional programming. It was introduced by Alonzo Church to formalise two key concepts when dealing with functions in mathematics and logic namely: function definition and function application. In this lecture, we build enough stuff to get a lambda calculus evaluator.

```
> module Lambda where
> import Data.List -- I need some standard list functions
```

We start with an example. Consider the squaring function, i.e. the function that maps x to x^2. In the notation of lambda calculus this is denoted as $\lambda x.x^2$. This is called a lambda abstraction. Apply a function f on an expression N is written as fN. The key rule in expression evaluation is the β-reduction: the expression $(\lambda x.M)N$ reduces under β-reduction to the expression M with N substituted in it. We now look at lambda calculus formally.

The goal of this lecture is to introduce basics of lambda calculus and on the way implement a small lambda calculus interpreter.

1 Abstract syntax

As with any other formal system we first define its abstract syntax. A lambda calculus expression is defined via the grammar

$$e := v | e_1 e_2 | \lambda x.e$$

Here e_1 and e_2 expressions themselves. We now capture this abstract syntax as a Haskell data type

```
> -- | The datatype that captures the lambda calculus expressions.
> data Expr = V    String       -- ^ A variable
>           | A    Expr   Expr  -- ^ functional application
>           | L    String Expr  -- ^ lambda abstraction
>           deriving Show
```

2 Free and bound variables

The notion of free and bound variables are fundamental to whole of mathematics. For example in the integral $\int \sin xydy$, the variable x occurs *free* where as the variables y occurs *bound* (to the corresponding $\int dy$). Clearly the value of the expression does *not* depend on the bound variable; in fact we can write the same integral as $\int \sin x t dt$.

In lambda calculus we say a variable occurs bound if it can be linked to a lambda abstraction. For example in the expression $\lambda x.xy$ the variable x is bound where as y occurs free. A variable can occur free as well as bound in an expression — consider x in $\lambda y.x(\lambda x.x)$.

Formally we can define the free variables of a lambda expression as follows.

$$FV(v) = \{v\}$$
$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$$
$$FV(\lambda x.e) = FV(e) \setminus \{x\}$$

We turn this into haskell code

```
> freeVar :: Expr -> [String]
> freeVar (V x  ) = [x]
> freeVar (A f e) = freeVar f ‘union‘ freeVar e
> freeVar (L x e) = delete x $ freeVar e
```

3 Variable substitution

We often need to substitute variables with other expressions. Since it is so frequent we give a notation for this. By $M[x := e]$, we mean the expression obtained by replacing all free occurrence of x in M by e. Let us code this up in haskell.

```
> subst :: Expr -> String -> Expr -> Expr
> subst var@(V y)   x e | y == x    = e
>                       | otherwise = var
> subst (A f a) x e                 = A (subst f x e) (subst a x e)
> subst lam@(L y a) x e | y == x    = lam
>                       | otherwise = L y (subst a x e)
```

4 Change of bound variables (α-reduction)

You are already familiar with this in mathematics. If we have an integral of the kind $\int xtdt$ we can rewrite it as $\int xydy$ by a change of variable. The same is true for lambda calculus. We say call the “reduction” $\lambda x.M \leftarrow \lambda t.M[x := t]$ as the α-reduction. However care should be taken when the new variable is chosen. Two pitfalls to avoid when performing α-reduction of the expression $\lambda x.M$ to $\lambda t.M[x := t]$ is

1. The variable t should not be free in M for otherwise by changing from x to t we have bound an otherwise free variable. For example if $M = t$ then $\lambda t.M[x = t]$ becomes $\lambda t.t$ which is clearly wrong.
2. If M has a free occurrence of x in the scope of a bound occurrence of t then we cannot perform change the bound variable x to t. For example consider $M = \lambda t.xt$. Then $\lambda t.M[x = t]$ will become $\lambda t.\lambda t.tt$ which is clearly wrong.

Clearly, one safe way to do α-reduction on $\lambda x.M$ is to use a fresh variable t, i.e. a variable that is neither free nor bound in M. We write a helper function to compute all the variables of a lambda calculus expression.

```
> varsOf :: Expr -> [String]
> varsOf (V x)   = [x]
> varsOf (A f e) = varsOf f `union` varsOf e
> varsOf (L x e) = varsOf e `union` [x]
```

We now give the code to perform a safe change of bound variables.

```
> alpha :: Expr -> [String] -> Expr
> alpha (A f e) vars                    = A (alpha f vars) (alpha e vars)
> alpha (L x e) vars | x `elem` vars     = L t $ alpha e' vars
>                    | otherwise        = L x $ alpha e  vars
>       where t  = fresh (varsOf e `union` vars)
>             e' = subst e x (V t)
> alpha  e       _                      = e
```

5 Function evaluation (β-reduction)

The way lambda calculus captures computation is through β reduction. We already saw what is β reduction. Under beta reduction, an expression $(\lambda x.M)N$ reduces to $M[x := N]$, where $M[x := N]$ denotes substitution of *free* occurrences of x by N. However, there is a chance that a free variable of N could become bound suddenly. For example consider N to be just y and M to be $\lambda y.xy$. Then reducing $(\lambda x.M)N$ to $M[x := N]$ will bind the free variable y in N.

We now give the code for β reduction. It performs one step of beta reduction that too if and only if the expression is of the form $(\lambda x.M)N$.

```
> beta :: Expr -> Expr
> beta (A (L x m) n) = carefulSubst m x n
> carefulSubst m x n = subst (alpha m $ freeVar n) x n
>
```

6 Generating Fresh variables

We saw that for our evaluator we needed a source of fresh variables. Our function fresh is given a set of variables and its task is to compute a variable name that does not belong to the list. We use diagonalisation, a trick first used by Cantor to prove that Real numbers are of strictly higher cardinality than integers.

```
> fresh :: [String] -> String
> fresh = foldl diagonalise "a"
>
> diagonalise  []        []                = "a" -- diagonalised any way
> diagonalise  []      (y:ys) | y == 'a'  = "b" -- diagonalise on this character
>                             | otherwise = "a"
> diagonalise  s        []                =  s  -- anyway s is differnt from t
> diagonalise s@(x:xs) (y:ys) | x /= y    = s   -- differs at this position anyway
>                             | otherwise = x : diagonalise xs ys
>
```

7 Exercise

1. Read up about β-normal forms. Write a function that converts a lambda calculus expression to its normal form if it exists. There are different evaluation strategies to get to β-normal form. Code them all up.
2. The use of `varOf` in α-reduction is an overkill. See if it can be improved.
3. Read about η-reduction and write code for it.