# Lecture 19: Nameless lambda expressions (DeBruijn); Intro to type systems

## CS4400 Programming Languages

As we have talked about before, the name of a bound variable is insignificant – as long as there is a consistency between the binder and the variable reference. This is called alpha-equivalence. Using strings as variables seems natural: it corresponds to the mathematical with variable names. However, as a computer representation this is inefficient: there are infinitely many ways to represent any lambda term: `λx. x`, `λy. y`, `λz. z`, `λzzzz. zzzz`, `λlongervariablename. longervariablename`, ...

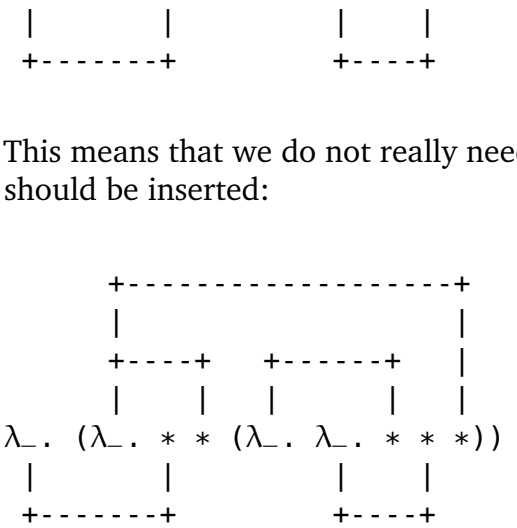
Moreover representing variable names as strings forces us to complicate the definition of substitution and define functions for obtaining fresh variable names. In an implementation, we would ideally want to do away with these complications. Sure, we could simply use natural numbers to represent variables and this would simplify picking a fresh variable name – e.g., by taking the maximum of all free variables and adding 1. We sill complicate the substitution definition and we still have the problem of multiple representations of alpha-equivalent lambda terms. There is another alternative.

When we look at a lambda abstraction:

$$\lambda x.\ (\lambda y.\ x\ y\ (\lambda x.\ \lambda z.\ x\ z\ y))$$

we really use the occurrence of $x$ in the binder as a *marker* and a variable reference as a reference back to the marker, that is, each variable can be viewed as referring back to the binder that bound it:

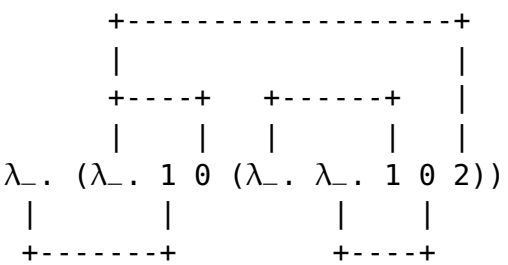
```
      +-------------------+
      |                   |
      +----+   +------+   |
      |    |   |      |   |
λx. (λy. x y (λx. λz. x z y))
```



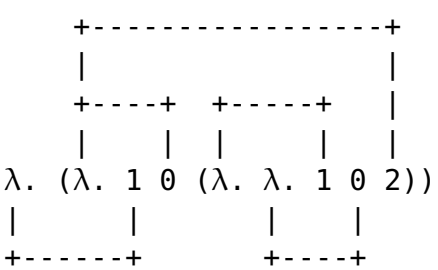

This means that we do not really need to use names to know where an argument value should be inserted:




Now the question is, how do we represent these connections without using names. The Dutch mathematician Nicolaas Govert de Bruijn had the idea that each variable reference should represent the number of binders between it and the binder that bound it. If there is no other binder between the reference and its binder, the count 0 and we can refer to the binder by that number. If there is one binder between, we refer to the variable's binder by 1, etc.




This leads to a simplification of the syntax: since we use do not need to mark binders using variables, lambdas do not carry any variable names:




Thus the syntax of Lambda expressions using de Bruijn indices is as follows:

```
<DLambda> ::= <Index>                 -- variable reference
            | <DLambda> <DLambda>     -- application is as before
            | λ. <DLambda>            -- lambda abstraction does refer to the boun
```

Haskell:

```haskell
data DLambda = DVar Integer
             | DApp DLambda DLambda
             | DLam DLambda
```

Here are a few more examples:

- Any identity function (`λx. x`, `λy. y`, ...) is `λ. 0`
- `λx. x x` is `λ. 0 0`
- Churchian for true and false is `λ. λ. 1` and `λ. λ. 0`, respectively
- Church numerals are `λ. λ. 0`, `λ. λ. 1 0`, `λ. λ. 1 (1 0)`, `λ. λ. 1 (1 (1 0))`
- The Y combinator, `λf. (λx. f (x x)) (λx. f (x x))` is `λ. (λ. 1 (0 0)) (λ. 1 (0 0))`

What is the advantage of using de Bruijn indices? It certainly isn't (human) readability. Maybe you have noticed, that for each alpha-equivalent term, there is only one representation. This is a major advantage when implementing lambda calculus, since we do not need to care about renaming of bound variables. Another advantage is that "environments" for evaluating lambda expressions are simplified – they are just stacks of values:

```haskell
data DValue = DClo DLambda [DValue]

eval :: [DValue] -> DLambda -> Maybe DValue
eval env (DVar i) | i < length env = Just (env!!i)   -- lookup the value
                  | otherwise = Nothing
eval env (DApp e1 e2) =
  case eval env e1 of
       Nothing -> Nothing
       Just (DClo e env') -> case eval env e2 of
                                  Nothing -> Nothing
                                  Just v2 -> eval (v2 : env') e
eval env (DLam e) = Just (DClo e env)
```