## Lambda abstraction

Since functions are first class entities, we should expect to find some notation in the language to create them from scratch. There are times when it is handy to just write a function "inline". The notation is:

```
\ x y -> x+y
```

This notation is based on the so-called "lambda-calculus".

## The $\lambda$-Calculus

- Invented by Alonzo Church in 1930s
- Intended as a form of logic
- Turned into a model of computation
- Not shown completely sound until early 70s !

## $\lambda$-Calculus: Syntax

Infinite set *Vars*, of variables:

$$u, v, x, y, z, \ldots, x_1, x_2, \ldots \quad \in \quad \mathit{Vars}$$

Well-formed $\lambda$-calculus expressions *LExpr* is the smallest set of strings matching the following syntax:

$$\begin{array}{rcl} M, N, \ldots \in \mathit{LExpr} & ::= & v \\ & | & (\lambda x \bullet M) \\ & | & (M\ N) \end{array}$$

Read: a $\lambda$-calculus expression is either (i) a variable ($v$); (ii) an abstraction of a variable from an expression ($\lambda x \bullet M$); or (iii) an application of one expression to another ($(M\ N)$).

## $\lambda$-Calculus: Free/Bound Variables

- A variable *occurrence* is *free* in an expression if it is not mentioned in an *enclosing abstraction*.
- A variable *occurrence* is *bound* in an expression if is mentioned in an *enclosing abstraction*.
- A variable *occurrence* is *binding* in an expression if it is the variable immediately after a $\lambda$.
- A variable is *free* in an expression if it occurs free in that expression.
- A variable is *bound* in an expression if it occurs bound in that expression.

A variable can be both free and bound in the same expression (how?)

## $\lambda$-Calculus: Computing Free/Bound Vars

We can define functions that return the sets of free and bound variables:

$$\begin{array}{rcl} \mathcal{FV} & : & LExpr \to \mathbb{P}\, Var \\ \mathcal{FV}(x) & \widehat{=} & \{x\} \\ \mathcal{FV}(\lambda x \bullet M) & \widehat{=} & \mathcal{FV}(M) \setminus \{x\} \\ \mathcal{FV}(M\ N) & \widehat{=} & \mathcal{FV}(M) \cup \mathcal{FV}(N) \end{array}$$

$$\begin{array}{rcl} \mathcal{BV} & : & LExpr \to \mathbb{P}\, Var \\ \mathcal{BV}(x) & \widehat{=} & \emptyset \\ \mathcal{BV}(\lambda x \bullet M) & \widehat{=} & \mathcal{BV}(M) \cup (\mathcal{FV}(M) \cap \{x\}) \\ \mathcal{BV}(M\ N) & \widehat{=} & \mathcal{BV}(M) \cup \mathcal{BV}(N) \end{array}$$

## $\lambda$-Calculus: $\alpha$-Renaming

We can change a binding variable and its bound instances provided we are careful not to make other free variables become bound.

$$\begin{array}{rcl} (\lambda x \bullet (\lambda y \bullet (x\ y))) & \xrightarrow{\alpha} & (\lambda u \bullet \lambda v \bullet (u\ v))) \\ (\lambda x \bullet (x\ y)) & \not\xrightarrow{\alpha} & (\lambda y \bullet (y\ y)) \\ & & \text{formerly free } y \text{ has been ``captured'' !} \end{array}$$

This process is called $\alpha$-Renaming or $\alpha$-Substitution, and leaves the meaning of a term unchanged.

## $\lambda$-Calculus: Substitution

We define the notion of substituting an expression $N$ for all free occurrences of $x$, in another expression $M$, written:

$$M[N/x]$$

$$(x\ (\lambda y \bullet (z\ y)))\ [\ (\lambda u \bullet u)\ /\ z\ ] \xrightarrow{\rho} (x\ (\lambda y \bullet ((\lambda u \bullet u)\ y)))$$

$$(x\ (\lambda y \bullet (z\ y)))\ [\ (\lambda u \bullet u)\ /\ y\ ] \xrightarrow{\rho} (x\ (\lambda y \bullet (z\ y)))$$

$y$ was not free anywhere

## $\lambda$-Calculus: Careful Substitution!

When doing (general) substitution $M[N/x]$, we need to avoid variable "capture" of free variables in $N$, by bindings in $M$:

$$(x\ (\lambda y \bullet (z\ y)))[(y\ x)/z] \not\xrightarrow{\rho} (x\ (\lambda y \bullet ((y\ x)\ y)))$$

If $N$ has free variables which are going to be inside an abstraction on those variables in $M$, then we need to $\alpha$-Rename the abstractions to something else first, and then substitute:

$$\begin{array}{rl} & (x\ (\lambda y \bullet (z\ y)))[(y\ x)/z] \\ \xrightarrow{\alpha} & (x\ (\lambda w \bullet (z\ w)))[(y\ x)/z] \\ \xrightarrow{\rho} & (x\ (\lambda w \bullet ((y\ x)\ w))) \end{array}$$

The Golden Rule: A substitution should never make a free occurrence of a variable become bound, or vice-versa.

## $\lambda$-Calculus: $\beta$-Reduction

We can now define the most important "move" in the $\lambda$-calculus, known as $\beta$-Reduction:

$$(\lambda x \bullet M)\ N \ \stackrel{\beta}{\rightarrow}\ M[N/x]$$

We define an expression of the form $(\lambda x \bullet M)\ N$ as a "($\beta-$)redex" (*red*ucible *ex*pression).

## $\lambda$-Calculus: Normal Form

An expression is in "Normal-Form" if it contains no redexes. The object of the exercise is to reduce an expression to its normal-form (*if it exists*).

$$\begin{aligned} & (\underline{(\lambda x \bullet (\lambda y \bullet (y\ x)))\ u})\ v) \\ \stackrel{\beta}{\rightarrow}\ & \underline{((\lambda y \bullet (y\ u))\ v)} \\ \stackrel{\beta}{\rightarrow}\ & (v\ u) \end{aligned}$$

Not all expressions have a normal form — e.g.:
$((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x)))$
What about:

$$(\ (\ (\lambda x \bullet (\lambda y \bullet y)))\ \ ((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x)))\ )\ \ w\ )\ \ ?$$

$$(\ (\ (\lambda x \bullet (\lambda y \bullet y)))\ \ ((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x)))\ )\ \ w\ )$$

- Do innermost redex first

$$\begin{aligned} & (\ (\ (\lambda x \bullet (\lambda y \bullet y)))\ \ (\underline{(\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x))}\ )\ \ w\ ) \\ \stackrel{\beta}{\rightarrow}\ & (\ (\ (\lambda x \bullet (\lambda y \bullet y)))\ \ ((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x)))\ )\ \ w\ ) \end{aligned}$$

  We can keep doing this forever!

- Do outermost redex first

$$\begin{aligned} & (\ \underline{(\ (\lambda x \bullet (\lambda y \bullet y)))\ \ ((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x)))\ }\ \ w\ ) \\ \stackrel{\beta}{\rightarrow}\ & (\ (\lambda y \bullet y)\ \ w\ ) \\ \stackrel{\beta}{\rightarrow}\ & w \end{aligned}$$

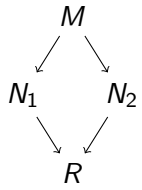
## $\lambda$-Calculus and Computability

- So What ? Why do we look at this weird calculus anyway?
- We can use it to encode booleans, numbers, and functions over same.
- In fact, we can encode any computable function this way!
- $\lambda$-Calculus is Turing-complete
  - or is it that Turing machines are Church-complete?
- It is one of a number of equivalent models of computation that emerged in the 1930s

## A Theorem: Church-Rosser

Question: how important is reduction order ?
The Church-Rosser theorem states:

> *If we can reduce $M$ to $N_1$ by one set of redex choices, and to $N_2$ by another, then there always exists a third value $R$ to which both $N_1$ and $N_2$ can be reduced.*




This third value may be one of $N_1$ or $N_2$, it need not be different.

## $\lambda$-Calculus: Reduction Order

The question is, if we have a choice of redexes, which should we reduce first ?
Answer: always the leftmost-outermost one (Normal Order Reduction).
*If* an expression has a normal form, then normal order reduction is *guaranteed* to find it.

## Lambda abstraction in Haskell

The Haskell notation is designed to reflect how it looks in lambda-calculus
Since these values are themselves functions, we just apply them to values to compute something

```
> (\x y -> x+y) 1 2
3
```

Essentially we can view Haskell as being the lambda-calculus with *LOTS* of syntactic sugar.