# Programing in λ-calculus

So far, only trivial functions:
identity, projections

Can we write interesting programs?

Let's start with data types

## Church Numerals

We choose some λ-terms to represent natural numbers

$$\bar{0} := \lambda f.\lambda x.x$$

$$\bar{1} := \lambda f.\lambda x.f\,x$$

$$\bar{2} := \lambda f.\lambda x.f\,(f\,x)$$

$$\bar{3} := \lambda f.\lambda x.f\,(f\,(f\,x)))$$

$$\vdots$$

$$\bar{n} := \lambda f.\lambda x.\underbrace{f\,(\ldots\,(f\,x)\ldots)}_{n \text{ times}}$$

$\bar{n}$ is an operator that

- when applied to a function $f$ and an argument $x$
- applies $f$ $n$ times to $x$.

We have assigned a different λ-term to each number

Can we define operations on them?

The successor function on Church numerals:

$$\text{succ} := \lambda n.\lambda f.\lambda x. f\,(n\,f\,x)$$

Let's see if it works:

$$\text{succ}\ \overline{2} \rightsquigarrow^{*} \overline{3}\ ?$$

$$\text{succ}\ \overline{2} = (\lambda n.\lambda f.\lambda x. f\,(n\,f\,x))\ \overline{2}$$
$$\rightsquigarrow \lambda f.\lambda x. f\,(\overline{2}\,f\,x)$$
$$= \lambda f.\lambda x. f\,((\lambda f.\lambda x. f\,(f\,x))\,f\,x)$$
$$\rightsquigarrow \lambda f.\lambda x. f\,((\lambda x. f\,(f\,x))\ x)$$
$$\rightsquigarrow \lambda f.\lambda x. f\,(f\,(f\,x))$$
$$= \overline{3}$$

Make sure you understand well how each step of reduction and substitution works

Other arithmetic operations:

$$\text{plus} := \lambda m.\lambda n.\lambda f.\lambda x.\ m\,f\,(n\,f\,x)$$

$$\text{mult} := \lambda m.\lambda n.\lambda f.\ m\,(n\,f)$$

$$\text{exp} := \lambda m.\lambda n.\ n\,m$$

isn't this amazing!

**Exercise:**
Verify that these definitions work
Do all reduction steps carefully

For example:

$\text{plus}\ \bar{2}\ \bar{3} \leadsto^* \bar{5}$

$\text{mult}\ \bar{2}\ \bar{3} \leadsto^* \bar{6}$

$\text{exp}\ \bar{2}\ \bar{3} \leadsto^* \bar{8}$

$\text{exp}\ \bar{3}\ \bar{2} \leadsto^* \bar{9}$

**Challenging Exercise:**

Try to define predecessor and subtraction: two λ-terms **pred** and **minus** such that

$\text{pred}\ \bar{3} \leadsto^* \bar{2}$

$\text{pred}\ \bar{0} \leadsto^* \bar{0}$

$\text{minus}\ \bar{5}\ \bar{2} \leadsto^* \bar{3}$

$\text{minus}\ \bar{2}\ \bar{5} \leadsto^* \bar{0}$

# Precise Definition of <u>Substitution</u>

## α-conversion:

The names of abstracted variables don't matter

$$\lambda x.x = \lambda y.y = \lambda z.z$$

But only occurrences <u>in scope</u> can be changed

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

in scope — bound variable

not in scope — unbound (free) variable

It is dangerous to use the same name for bound and free variables. Try to avoid it

If free and bound variables have the same name, rename the bound one.

Similarly:
Avoid binding the same variable more than once.

## Substitution (with variable capture)

$$(\lambda y.M)[x{:=}N] \overset{?}{=} \lambda y.M[x{:=}N]$$

Before substituting rename bound variables so they are unique

If $y$ occurs free in $N$, this is wrong the free variable gets "captured" by the abstraction

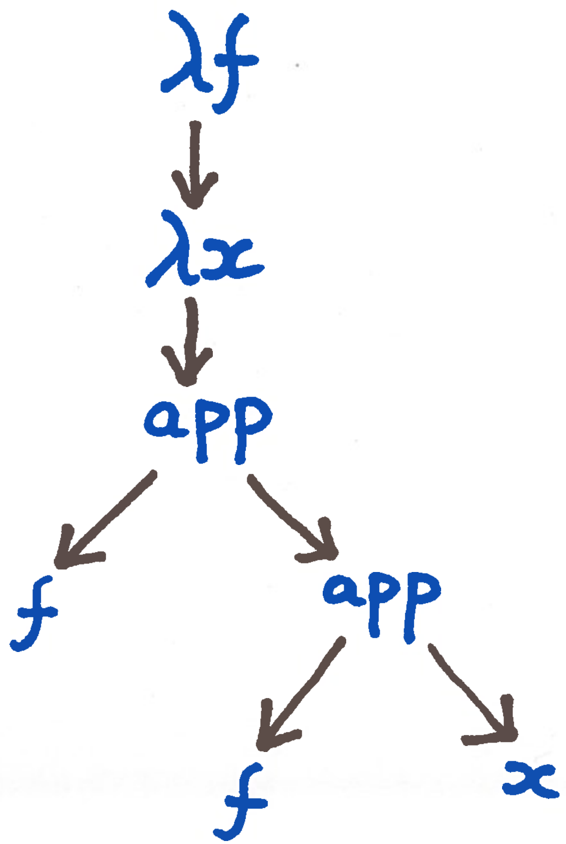
## Abstract Syntax Trees

Alternative Representation of $\lambda$-terms as trees with

nodes = abstractions/applications

leaves = variables

Example: $\bar{2} = \lambda f.\lambda x.f(f\,x)$

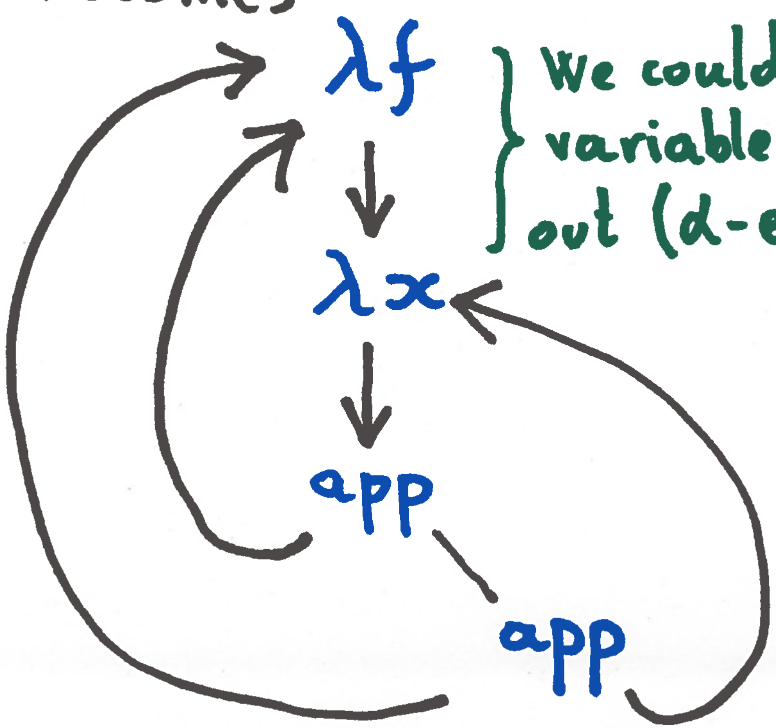


More efficient representation:

## Term Graphs

- Bound variable occurrences point to their abstractions
- A subgraph can have many incoming edges (sharing)

$\overline{2}$ becomes

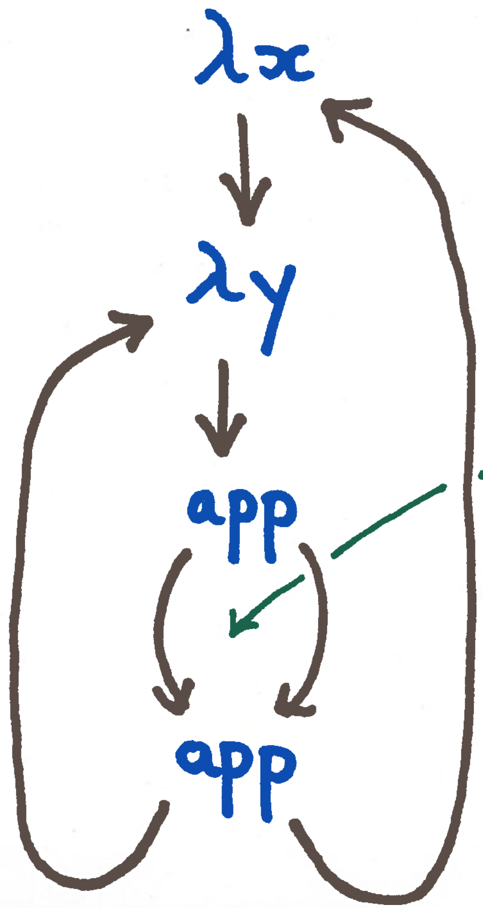



We could leave variable names out (α-equiv.)

Term with sharing:

$$\lambda x.\lambda y.\underbrace{(y\,x)\,(y\,x)}$$

two occurrences of the same term




two references to the same subgraph

**Attention:**
**When reducing we may need to duplicate**

# Free and Bound Variables

**Free occurrences of a variable**

those that are not in scope of a $\lambda$-abstraction for that variable

Definition of set of free variables (FV) by structural recursion

$$FV(x) = \{x\}$$

$$FV(\lambda x.t) = FV(t) \setminus \{x\}$$

$$FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$$

# Precise definition of substitution

(also by structural recursion)

$$x[x:=N] = N$$

$$y[x:=N] = y \quad \text{if } y \neq x$$

$$(\lambda y.M)[x:=N] = \begin{cases} \lambda y.M & \text{if } y = x \\ \lambda y.M[x:=N] & \text{if } y \neq x,\ y \notin FV(N) \end{cases}$$

$$(M_1\ M_2)[x:=N] = M_1[x:=N]\ M_2[x:=N]$$

In the third case

if $y \neq x$ and $y \in FV(N)$ use $\alpha$-conversion to change $y$ to a variable $z$ such that $z \neq x$, $z \notin FV(M)$, $z \notin FV(N)$

Example:

$$(\lambda x. y\, x)[y := x\, x]$$
$$\equiv_\alpha$$
$$(\lambda z. y\, z)[y := x\, x]$$
$$=$$
$$\lambda z. (x\, x) z$$

## Barendregt Convention

In a term

- A variable name is bound by at most one $\lambda$-abstraction
- Free variables and Bound variables have different names