

# The Untyped Lambda Calculus

Manuel Eberl  
eberlm@cs.tum.edu

August 20, 2011

# What is the Lambda calculus?

- developed by Alonzo Church in the 1930s
- a formal system for definition and analysis of functions
- as powerful as the Turing machine
- forms the basis of functional programming languages

# Basic idea

Everything is a function and functions are defined and applied in one expression

## Example

Consider these functions:

- *sqr*: expects a number and returns its square
- *twice*: expects a function  $f$  and returns  $f \circ f$
- $sqr\ 3 = 9$

# Basic idea

Everything is a function and functions are defined and applied in one expression

## Example

Consider these functions:

- *sqr*: expects a number and returns its square
- *twice*: expects a function  $f$  and returns  $f \circ f$
- $sqr\ 3 = 9$
- $(twice\ sqr)\ 3 = sqr\ (sqr\ 3) = 81$

# Formal Definition

The Lambda Calculus has three kinds of expressions:

## Definition

Lambda expressions are:

**Variables** e.g.  $a, b, c, \dots$

**Abstractions** e.g.  $\lambda x. T$  where  $x$  is a variable and  $T$  is a Lambda expression

**Applications** e.g.  $S T$  where  $S$  and  $T$  are Lambda expressions

Additionally: parentheses for grouping

# Abstractions

## Abstraction

$\lambda x. T$  - defines a function that maps  $x$  to  $T$ .  $T$  is a Lambda expression which usually contains  $x$  (but not necessarily)

## Example

- $\lambda x. x$  (the identity function *id*)
- $\lambda x. a$  (a function that always returns  $a$ )

# Applications

## Application

$S$   $T$  - applies the function  $S$  to the expression  $T$ .

## Definition ( $\beta$ Reduction)

If  $S$  is of the shape  $(\lambda x. R) T$  (so-called redex for reducible expression): Reduction by replacing all  $x$  in  $R$  by  $T$

$$(\lambda x. x y x) T \longrightarrow T y T$$

## Example

$$\begin{aligned} & (\lambda x. 42 x y x 1337) \text{wuppd}i \\ & \quad \downarrow \beta \\ & 42 \text{wuppd}i y \text{wuppd}i 1337 \end{aligned}$$

## Examples

- $id\ a$

## Examples

- $id\ a$  - yields  $a$
- $id\ id$

# Applications

## Examples

- *id a* - yields *a*
- *id id* - yields *id*
- *sqr 3*

# Applications

## Examples

- *id a* - yields *a*
- *id id* - yields *id*
- *sqr 3* - yields 9
- *twice sqr*

# Applications

## Examples

- *id a* - yields *a*
- *id id* - yields *id*
- *sqr 3* - yields 9
- *twice sqr* - yields a “to the 4<sup>th</sup> power” function
- *twice sqr 3*

## Examples

- *id a* - yields *a*
- *id id* - yields *id*
- *sqr 3* - yields 9
- *twice sqr* - yields a “to the 4<sup>th</sup> power” function
- *twice sqr 3* - yields 81

Caution: applications are left-associative.  $f f x$  means  $(f f) x$ , not  $f (f x)$

## Functions with more than one parameter - Currying

Problem: only one parameter per abstraction permitted. How to define *add*  $a\ b = a + b$ ?

## Functions with more than one parameter - Currying

Problem: only one parameter per abstraction permitted. How to define  $add\ a\ b = a + b$ ?

Solution: nesting functions -  $add(a)$  returns a function  $add_a(b)$  with constant  $a$ , which maps  $b$  to  $a + b$ .  
This nesting is called “Currying”.

# Functions with more than one parameter - Currying

Problem: only one parameter per abstraction permitted. How to define  $add\ a\ b = a + b$ ?

Solution: nesting functions -  $add(a)$  returns a function  $add_a(b)$  with constant  $a$ , which maps  $b$  to  $a + b$ .

This nesting is called “Currying”.

## Example 1 - Addition

- Let  $+$  be a function that adds its two parameters

# Functions with more than one parameter - Currying

Problem: only one parameter per abstraction permitted. How to define  $add\ a\ b = a + b$ ?

Solution: nesting functions -  $add(a)$  returns a function  $add_a(b)$  with constant  $a$ , which maps  $b$  to  $a + b$ .

This nesting is called “Currying”.

## Example 1 - Addition

- Let  $+$  be a function that adds its two parameters
- $+ 4 = +_4$  ist an “Increase by 4” function

# Functions with more than one parameter - Currying

Problem: only one parameter per abstraction permitted. How to define  $add\ a\ b = a + b$ ?

Solution: nesting functions -  $add(a)$  returns a function  $add_a(b)$  with constant  $a$ , which maps  $b$  to  $a + b$ .

This nesting is called “Currying”.

## Example 1 - Addition

- Let  $+$  be a function that adds its two parameters
- $+ 4 = +_4$  ist an “Increase by 4” function
- $+ 4 2 = +_{4,2}$  is equivalent to 6

## Functions with more than one parameter - Currying

### Example 2- reverse application function

Definition: *reverse*  $x$   $y$  returns  $y$   $x$

*reverse* :=  $\lambda x. \lambda y. y$   $x$

## Functions with more than one parameter - Currying

### Example 2- reverse application function

Definition: *reverse*  $x$   $y$  returns  $y$   $x$

$reverse := \lambda x. \lambda y. y$   $x$

$reverse$   $a$   $b \equiv$

## Functions with more than one parameter - Currying

### Example 2- reverse application function

Definition: *reverse*  $x$   $y$  returns  $y$   $x$

$reverse := \lambda x. \lambda y. y x$

$reverse\ a\ b \equiv \underline{(\lambda x. \lambda y. y\ x)}\ a\ b$

## Functions with more than one parameter - Currying

### Example 2- reverse application function

Definition: *reverse*  $x$   $y$  returns  $y$   $x$

$reverse := \lambda x. \lambda y. y x$

$reverse\ a\ b \equiv \underline{(\lambda x. \lambda y. y\ x)\ a}\ b \equiv reverse_a\ b \equiv \underline{(\lambda y. y\ a)\ b}$

## Functions with more than one parameter - Currying

### Example 2- reverse application function

Definition: *reverse*  $x$   $y$  returns  $y$   $x$

$reverse := \lambda x. \lambda y. y$   $x$

$reverse$   $a$   $b \equiv \underline{(\lambda x. \lambda y. y$   $x)}$   $a$   $b \equiv reverse_a$   $b \equiv \underline{(\lambda y. y$   $a)}$   $b \equiv b$   $a$

Shorthand for nested functions: Instead of  $\lambda x. \lambda y. \lambda z$   $T$  write  $\lambda xyz.$   $T$ .

Example:  $reverse := \lambda xy. y$   $x$

# Free variables and bound variables

A variable in an abstraction is called bound if it was defined by a lambda.

Non-bound variables are free. (cf.  $\exists$  and  $\forall$ )

## Example

- $\lambda xy. x y$  -  $x$  and  $y$  are bound
- $\lambda x. x y$  -  $x$  is bound,  $y$  is free
- $(\lambda x. x) x$  - inner  $x$  is bound, outer  $x$  is free
- $\lambda x. \lambda x. x$  -  $x$  is bound and “hides” the outer  $x$

## $\alpha$ conversion

What is the difference between  $\lambda x.x$  and  $\lambda y.y$ ?

## $\alpha$ conversion

What is the difference between  $\lambda x. x$  and  $\lambda y. y$ ?

Answer: none at all! Names of bound variables are irrelevant.

Renaming bound variables is called  $\alpha$  conversion.

## $\alpha$ conversion

What is the difference between  $\lambda x. x$  and  $\lambda y. y$ ?

Answer: none at all! Names of bound variables are irrelevant.

Renaming bound variables is called  $\alpha$  conversion.

### Definition ( $\alpha$ conversion)

A  $\lambda$  abstraction  $\lambda x. A$  is equivalent to  $\lambda y. A[x := y]$

where  $A[x := y]$  means: all occurrences of  $x$  in  $A$  are replaced with  $y$  (but: take variable hiding into account)

## $\alpha$ conversion

What is the difference between  $\lambda x. x$  and  $\lambda y. y$ ?

Answer: none at all! Names of bound variables are irrelevant.  
Renaming bound variables is called  $\alpha$  conversion.

### Definition ( $\alpha$ conversion)

A  $\lambda$  abstraction  $\lambda x. A$  is equivalent to  $\lambda y. A[x := y]$

where  $A[x := y]$  means: all occurrences of  $x$  in  $A$  are replaced with  $y$  (but: take variable hiding into account)

Expressions that can be converted to one another by  $\alpha$  conversion are called  $\alpha$ -equivalent.

## Collision of variable names

**Caution:** When replacing variables or applying  $\beta$  reduction, variables defined on deeper levels must not be overwritten!

Example - collision after  $\alpha$ -Konversion

$\lambda xy. x$  must not be  $\alpha$  converted to  $\lambda y. \lambda y. y$

## Collision of variable names

**Caution:** When replacing variables or applying  $\beta$  reduction, variables defined on deeper levels must not be overwritten!

Example - collision after  $\alpha$ -Konversion

$\lambda xy. x$  must not be  $\alpha$  converted to  $\lambda y. \lambda y. y$

Solution: replace colliding variable as well:

$\lambda xy. x \equiv \lambda xz. x \equiv \lambda yz. y$

## Collision of variable names

**Caution:** When replacing variables or applying  $\beta$  reduction, variables defined on deeper levels must not be overwritten!

### Example - collision after $\alpha$ -Konversion

$\lambda xy. x$  must not be  $\alpha$  converted to  $\lambda y. \lambda y. y$

Solution: replace colliding variable as well:

$\lambda xy. x \equiv \lambda xz. x \equiv \lambda yz. y$

### Example - collision after $\beta$ reduction

$(\lambda ab. a) b$  - first  $b$  bound, second  $b$  free.

## Collision of variable names

**Caution:** When replacing variables or applying  $\beta$  reduction, variables defined on deeper levels must not be overwritten!

### Example - collision after $\alpha$ -Konversion

$\lambda xy. x$  must not be  $\alpha$  converted to  $\lambda y. \lambda y. y$

Solution: replace colliding variable as well:

$\lambda xy. x \equiv \lambda xz. x \equiv \lambda yz. y$

### Example - collision after $\beta$ reduction

$(\lambda ab. a) b$  - first  $b$  bound, second  $b$  free.

After  $\beta$  reduction:

$\lambda b. b$  - bound  $b$  is returned - different result!

# Collision of variable names

**Caution:** When replacing variables or applying  $\beta$  reduction, variables defined on deeper levels must not be overwritten!

## Example - collision after $\alpha$ -Konversion

$\lambda xy. x$  must not be  $\alpha$  converted to  $\lambda y. \lambda y. y$

Solution: replace colliding variable as well:

$\lambda xy. x \equiv \lambda xz. x \equiv \lambda yz. y$

## Example - collision after $\beta$ reduction

$(\lambda ab. a) b$  - first  $b$  bound, second  $b$  free.

After  $\beta$  reduction:

$\lambda b. b$  - bound  $b$  is returned - different result!  $\Rightarrow$  Rename bound variable:  $\lambda b'. b$

# Data types?

Problem: no data types such as numbers, booleans, lists, strings - there are only functions. But: Lambda calculus is Turing complete, so it has to be possible to e.g. do calculations with numbers. But how?

Solution: encode all data types as functions!

## Definition of true and false

*true* and *false* as functions of two parameters:  $true\ a\ b \equiv a$  and  $false\ a\ b \equiv b$ .

### Definition

$true := \lambda ab. a$   
 $false := \lambda ab. b$

Booleans can be used to make “branches”.

“if  $A$  then  $B$  else  $C$ ” where  $A$  is a boolean can be expressed as  $A\ B\ C$ .

# Boolean operations

## Boolean operations

- *not* - **if** *a* **then** *false* **else** *true*; -

# Boolean operations

## Boolean operations

- *not* - **if  $a$  then  $false$  else  $true$** ; -  $\lambda a. a \text{ false } true$

# Boolean operations

## Boolean operations

- *not* - **if  $a$  then  $false$  else  $true$** ; -  $\lambda a. a \text{ false } true$
- *and* - **if  $a$  then  $b$  else  $false$** ; -

# Boolean operations

## Boolean operations

- *not* - **if  $a$  then  $false$  else  $true$** ; -  $\lambda a. a \ false \ true$
- *and* - **if  $a$  then  $b$  else  $false$** ; -  $\lambda a. \lambda b. a \ b \ false$

# Boolean operations

## Boolean operations

- *not* - **if  $a$  then  $false$  else  $true$** ; -  $\lambda a. a \ false \ true$
- *and* - **if  $a$  then  $b$  else  $false$** ; -  $\lambda a. \lambda b. a \ b \ false$
- *or* - **if  $a$  then  $true$  else  $b$** ; -

# Boolean operations

## Boolean operations

- *not* - **if  $a$  then  $false$  else  $true$** ; -  $\lambda a. a \ false \ true$
- *and* - **if  $a$  then  $b$  else  $false$** ; -  $\lambda a. \lambda b. a \ b \ false$
- *or* - **if  $a$  then  $true$  else  $b$** ; -  $\lambda a. \lambda b. a \ true \ b$

# Boolean operations

## Boolean operations

- *not* - **if  $a$  then  $false$  else  $true$** ; -  $\lambda a. a \ false \ true$
- *and* - **if  $a$  then  $b$  else  $false$** ; -  $\lambda a. \lambda b. a \ b \ false$
- *or* - **if  $a$  then  $true$  else  $b$** ; -  $\lambda a. \lambda b. a \ true \ b$
- *xor* - **if  $a$  then  $not \ b$  else  $b$** ; -

# Boolean operations

## Boolean operations

- *not* - **if  $a$  then  $false$  else  $true$** ; -  $\lambda a. a \text{ false } true$
- *and* - **if  $a$  then  $b$  else  $false$** ; -  $\lambda a. \lambda b. a \text{ } b \text{ } false$
- *or* - **if  $a$  then  $true$  else  $b$** ; -  $\lambda a. \lambda b. a \text{ } true \text{ } b$
- *xor* - **if  $a$  then  $not \text{ } b$  else  $b$** ; -  $\lambda a. \lambda b. a \text{ } (not \text{ } b) \text{ } b$

## Definition of pairs of values

Useful for more complex calculations and data structures: encoding two values (expressions) in a pair

## Definition of pairs of values

Useful for more complex calculations and data structures: encoding two values (expressions) in a pair

A pair  $(a, b)$  is a function that expects a function  $z$  and applies  $z$  to  $a$  and  $b$

## Definition of pairs of values

Useful for more complex calculations and data structures: encoding two values (expressions) in a pair

A pair  $(a, b)$  is a function that expects a function  $z$  and applies  $z$  to  $a$  and  $b$

### Definition

A pair  $(a, b)$  is encoded as:  $\lambda z. z a b$

## Definition of pairs of values

Useful for more complex calculations and data structures: encoding two values (expressions) in a pair

A pair  $(a, b)$  is a function that expects a function  $z$  and applies  $z$  to  $a$  and  $b$

### Definition

A pair  $(a, b)$  is encoded as:  $\lambda z. z a b$

Accessing pair elements with *first* and *second*

### Definition

$first := \lambda p. p \ true$

$second := \lambda p. p \ false$

(cf. Definition of *true* and *false*)

# Examples of pairs

## Beispiele

Let  $p$  be the pair  $(Karl, Ranseier)$ .

$\Rightarrow p := \lambda z. z \text{ Karl Ranseier}$

$first\ p = p\ true = Karl$

$second\ p = p\ false = Ranseier$

## Definition of Church numerals

How can we encode numbers from  $\mathbb{N}_0$ ?

## Definition of Church numerals

How can we encode numbers from  $\mathbb{N}_0$ ?

Solution: 0 is a function that applies another function 0 times, 1 applies it 1 times and so on

## Definition of Church numerals

How can we encode numbers from  $\mathbb{N}_0$ ?

Solution: 0 is a function that applies another function 0 times, 1 applies it 1 times and so on

### Definition

The Church numeral for the number  $n$  maps  $f$  to  $f^n$ .

# Definition of Church numerals

How can we encode numbers from  $\mathbb{N}_0$ ?

Solution: 0 is a function that applies another function 0 times, 1 applies it 1 times and so on

## Definition

The Church numeral for the number  $n$  maps  $f$  to  $f^n$ .

$$\Rightarrow n \hat{=} \lambda f x. \underbrace{f (f \dots (f x) \dots)}_{n \text{ mal}} = \lambda f x. f^n x$$

# Definition of Church numerals

How can we encode numbers from  $\mathbb{N}_0$ ?

Solution: 0 is a function that applies another function 0 times, 1 applies it 1 times and so on

## Definition

The Church numeral for the number  $n$  maps  $f$  to  $f^n$ .

$$\Rightarrow n \hat{=} \lambda f x. \underbrace{f (f \dots (f x) \dots)}_{n \text{ mal}} = \lambda f x. f^n x$$

## Examples

- $0 := \lambda f x. x$  (maps all functions to *id*)
- $1 := \lambda f x. f x$
- $2 := \lambda f x. f (f x)$  ( $\equiv$  *twice*)
- ...

## Arithmetic operations - Addition

Successor function: *succ*

Maps  $n$  to  $n + 1$ , i.e. *succ*  $n$  applies  $f$  once more than  $n$ .

## Arithmetic operations - Addition

Successor function: *succ*

Maps  $n$  to  $n + 1$ , i.e. *succ*  $n$  applies  $f$  once more than  $n$ .

$succ := \lambda nfx. f (n f x)$

## Arithmetic operations - Addition

Successor function: *succ*

Maps  $n$  to  $n + 1$ , i.e. *succ*  $n$  applies  $f$  once more than  $n$ .

$succ := \lambda n f x. f (n f x)$

Addition:  $+$

In order to add  $m$  and  $n$ : put  $n f$  as parameter into  $m f$ :

## Arithmetic operations - Addition

Successor function: *succ*

Maps  $n$  to  $n + 1$ , i.e. *succ*  $n$  applies  $f$  once more than  $n$ .

$succ := \lambda n f x. f (n f x)$

Addition:  $+$

In order to add  $m$  and  $n$ : put  $n f$  as parameter into  $m f$ :

$(m + n) \hat{=} \lambda f x. f^m (f^n x)$

## Arithmetic operations - Addition

Successor function: *succ*

Maps  $n$  to  $n + 1$ , i.e. *succ*  $n$  applies  $f$  once more than  $n$ .

$succ := \lambda n f x. f (n f x)$

Addition:  $+$

In order to add  $m$  and  $n$ : put  $n f$  as parameter into  $m f$ :

$(m + n) \hat{=} \lambda f x. f^m (f^n x)$

$\Rightarrow + := \lambda m n f x. m f (n f x)$

## Arithmetic operations - Addition

Example: *succ*

$\textit{succ } 1 \equiv \textit{succ } (\lambda f x. f x) \equiv$

## Arithmetic operations - Addition

Example: *succ*

$$\textit{succ } 1 \equiv \textit{succ } (\lambda f x. f x) \equiv \underline{(\lambda n f x. f (n f x))} \underline{(\lambda f x. f x)} \equiv$$

## Arithmetic operations - Addition

Example: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda f x. f x) \equiv (\lambda n f x. f (n f x)) (\lambda f x. f x) \equiv \\ &\lambda f x. f ((\lambda f' x'. f' x') \underline{f} \underline{x}) \equiv \end{aligned}$$

## Arithmetic operations - Addition

Example: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda f x. f \ x) \equiv (\lambda n f x. f \ (n \ f \ x)) \ (\lambda f x. f \ x) \equiv \\ &\lambda f x. f \ (\underline{(\lambda f' x'. f' \ x') \ f \ x}) \equiv \lambda f x. f \ (f \ x) = 2 \end{aligned}$$

## Arithmetic operations - Addition

Example: *succ*

$$\text{succ } 1 \equiv \text{succ } (\lambda f x. f x) \equiv (\lambda n f x. f (n f x)) (\lambda f x. f x) \equiv \lambda f x. f ((\lambda f' x'. f' x') f x) \equiv \lambda f x. f (f x) = 2$$

Example:  $+$

$$+ 2 3 \equiv (\lambda m n f x. m f (n f x)) 2 3 \equiv$$

## Arithmetic operations - Addition

Example: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda f x. f x) \equiv (\lambda n f x. f (n f x)) (\lambda f x. f x) \equiv \\ &\lambda f x. f ((\lambda f' x'. f' x') f x) \equiv \lambda f x. f (f x) = 2 \end{aligned}$$

Example:  $+$

$$\begin{aligned} + 2 3 &\equiv (\lambda m n f x. m f (n f x)) 2 3 \equiv \\ &\equiv \lambda f x. 2 f (3 f x) \equiv \end{aligned}$$

## Arithmetic operations - Addition

Example: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda f x. f x) \equiv (\lambda n f x. f (n f x)) (\lambda f x. f x) \equiv \\ &\lambda f x. f ((\lambda f' x'. f' x') f x) \equiv \lambda f x. f (f x) = 2 \end{aligned}$$

Example: *+*

$$\begin{aligned} + 2 3 &\equiv (\lambda m n f x. m f (n f x)) 2 3 \equiv \\ &\equiv \lambda f x. 2 f (3 f x) \equiv \\ &\equiv \lambda f x. (\lambda f' x'. f' (f' x')) f ((\lambda f' x'. f' (f' (f' x')))) f x \equiv \end{aligned}$$

## Arithmetic operations - Addition

Example: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda f x. f x) \equiv (\lambda n f x. f (n f x)) (\lambda f x. f x) \equiv \\ &\lambda f x. f ((\lambda f' x'. f' x') \underline{f} \underline{x}) \equiv \lambda f x. f (f x) = 2 \end{aligned}$$

Example:  $+$

$$\begin{aligned} + 2 3 &\equiv (\lambda m n f x. m f (n f x)) \underline{2} \underline{3} \equiv \\ &\equiv \lambda f x. \underline{2} f (\underline{3} f x) \equiv \\ &\equiv \lambda f x. (\lambda f' x'. f' (f' x')) \underline{f} ((\lambda f' x'. f' (f' (f' x')))) \underline{f} \underline{x}) \equiv \\ &\equiv \lambda f x. (\lambda x'. f (f x')) (f (f (f x))) \equiv \end{aligned}$$

## Arithmetic operations - Addition

Example: *succ*

$$\begin{aligned} \text{succ } 1 &\equiv \text{succ } (\lambda f x. f x) \equiv (\lambda n f x. f (n f x)) (\lambda f x. f x) \equiv \\ &\lambda f x. f ((\lambda f' x'. f' x') \underline{f} \underline{x}) \equiv \lambda f x. f (f x) = 2 \end{aligned}$$

Example: *+*

$$\begin{aligned} + 2 3 &\equiv (\lambda m n f x. m f (n f x)) \underline{2} \underline{3} \equiv \\ &\equiv \lambda f x. \underline{2} f (\underline{3} f x) \equiv \\ &\equiv \lambda f x. (\lambda f' x'. f' (f' x')) \underline{f} ((\lambda f' x'. f' (f' (f' x')))) \underline{f} \underline{x}) \equiv \\ &\equiv \lambda f x. (\lambda x'. f (f x')) (f (f (f x))) \equiv \\ &\equiv \lambda f x. f (f (f (f (f x)))) = 5 \end{aligned}$$

# Arithmetic operations - Multiplication

## Multiplication: \*

For numbers  $m$  and  $n$ : what happens when  $n$  is applied to  $f$  and  $m$  is applied to the result?

# Arithmetic operations - Multiplication

## Multiplication: $*$

For numbers  $m$  and  $n$ : what happens when  $n$  is applied to  $f$  and  $m$  is applied to the result?

$\Rightarrow n f$  is  $f$  chained  $n$  times. Therefore,  $m (n f)$  is  $f$  chained  $m \cdot n$  times.

# Arithmetic operations - Multiplication

## Multiplication: $*$

For numbers  $m$  and  $n$ : what happens when  $n$  is applied to  $f$  and  $m$  is applied to the result?

$\Rightarrow n f$  is  $f$  chained  $n$  times. Therefore,  $m (n f)$  is  $f$  chained  $m \cdot n$  times.

$\Rightarrow * := \lambda mnf. m (n f)$

## Arithmetic operations - Subtraction

We need the opposite of *succ*: a predecessor function.  
0 has no predecessor, therefore saturated subtraction:  
 $pred(n) := n \dot{-} 1$ , i.e. predecessor of 0 defined as 0.

## Arithmetic operations - Subtraction

We need the opposite of *succ*: a predecessor function.  
0 has no predecessor, therefore saturated subtraction:  
 $pred(n) := n \dot{-} 1$ , i.e. predecessor of 0 defined as 0.

Predecessor function: *pred*

Helper function  $\Phi$ , that maps  $p := (a, b)$  to  $(a + 1, a)$ :

## Arithmetic operations - Subtraction

We need the opposite of *succ*: a predecessor function.  
0 has no predecessor, therefore saturated subtraction:  
 $pred(n) := n \dot{-} 1$ , i.e. predecessor of 0 defined as 0.

Predecessor function: *pred*

Helper function  $\Phi$ , that maps  $p := (a, b)$  to  $(a + 1, a)$ :  
 $\Phi := \lambda pz. z (succ (first p)) (first p)$

## Arithmetic operations - Subtraction

We need the opposite of *succ*: a predecessor function.  
0 has no predecessor, therefore saturated subtraction:  
 $pred(n) := n \dot{-} 1$ , i.e. predecessor of 0 defined as 0.

Predecessor function: *pred*

Helper function  $\Phi$ , that maps  $p := (a, b)$  to  $(a + 1, a)$ :  
 $\Phi := \lambda p z. z (succ (first p)) (first p)$

$\Phi$   $n$  applied to  $(0, 0)$   $n$  times yields:  $(n, n \dot{-} 1)$   
 $\Rightarrow pred := \lambda n. second(n \Phi (\lambda z. z 0 0))$

## Arithmetic operations - Subtraction

Predecessor of 0 is 0.

⇒ Only saturated subtraction possible:  $m \dot{-} n = \max(m - n, 0)$

## Arithmetic operations - Subtraction

Predecessor of 0 is 0.

⇒ Only saturated subtraction possible:  $m \dot{-} n = \max(m - n, 0)$

Saturated subtraction:  $\dot{-}$

For  $m \dot{-} n$ : predecessor function *pred* is applied *n* times to *m*.

⇒  $\dot{-} := \lambda mn. n \text{ pred } m$

Negative numbers can be expressed in Lambda calculus. (e.g. pair of Church numeral and sign boolean)

# Comparison operators

Most important comparison: test if number is 0

## Comparison operators

Most important comparison: test if number is 0

*iszero*

Helper function  $h$ , that always returns *false*:  $h := \lambda x. \text{false}$

## Comparison operators

Most important comparison: test if number is 0

*iszero*

Helper function  $h$ , that always returns *false*:  $h := \lambda x. \text{false}$

$h^n = id$  if  $n = 0$ ; if  $n \neq 0$ ,  $h^n$  returns *false*

## Comparison operators

Most important comparison: test if number is 0

*iszero*

Helper function  $h$ , that always returns *false*:  $h := \lambda x. false$

$h^n = id$  if  $n = 0$ ; if  $n \neq 0$ ,  $h^n$  returns *false*

$\Rightarrow n h true$  is *true* iff  $n = 0$ , *false* otherwise.

$\Rightarrow iszero := \lambda n. n (\lambda x. false) true$

# Comparison operators

Less or equal:  $\leq$

$$m \leq n \Leftrightarrow m - n \leq 0 \Leftrightarrow m \dot{-} n = 0.$$

$$\Rightarrow \leq := \lambda mn. \text{iszero } (\dot{-} m n)$$

# Comparison operators

Less or equal:  $\leq$

$$m \leq n \Leftrightarrow m - n \leq 0 \Leftrightarrow m \dot{-} n = 0.$$

$$\Rightarrow \leq := \lambda mn. \text{iszero } (\dot{-} \ m \ n)$$

Greater or equal:  $\geq$

$$m \geq n \Leftrightarrow n - m \leq 0 \Leftrightarrow n \dot{-} m = 0.$$

$$\Rightarrow \geq := \lambda mn. \text{iszero } (\dot{-} \ n \ m)$$

# Comparison operators

Less or equal:  $\leq$

$$m \leq n \Leftrightarrow m - n \leq 0 \Leftrightarrow m \dot{-} n = 0.$$

$$\Rightarrow \leq := \lambda mn. \text{iszero } (\dot{-} \ m \ n)$$

Greater or equal:  $\geq$

$$m \geq n \Leftrightarrow n - m \leq 0 \Leftrightarrow n \dot{-} m = 0.$$

$$\Rightarrow \geq := \lambda mn. \text{iszero } (\dot{-} \ n \ m)$$

Equal:  $=$

$$\Rightarrow = := \lambda mn. \text{and } (\leq \ m \ n) \ (\geq \ m \ n)$$

# Comparison operators

Less/greater:  $<$  und  $>$

$$m < n \Leftrightarrow \neg(m \geq n)$$

$$\Rightarrow < := \lambda mn. \text{not } (\geq m n)$$

$$m > n \Leftrightarrow \neg(m \leq n)$$

$$\Rightarrow > := \lambda mn. \text{not } (\leq m n)$$

# Recursion

“Real” recursions with termination conditions are somewhat complicated in Lambda calculus

“Real” recursions with termination conditions are somewhat complicated in Lambda calculus

Example - the factorial *fac*

$$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot fac(n - 1) & \text{otherwise} \end{cases}$$

“Real” recursions with termination conditions are somewhat complicated in Lambda calculus

Example - the factorial *fac*

$$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot fac(n - 1) & \text{otherwise} \end{cases}$$

Intuitive solution:  $fac := \lambda n. (iszero\ n)\ 1\ (*\ n\ (fac\ (pred\ n)))$

# Recursion

“Real” recursions with termination conditions are somewhat complicated in Lambda calculus

Example - the factorial *fac*

$$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot fac(n - 1) & \text{otherwise} \end{cases}$$

Intuitive solution:  $fac := \lambda n. (iszero\ n)\ 1\ (*\ n\ (fac\ (pred\ n)))$

But: *fac* has to be plugged into *fac* again, expression grows infinitely

## Example - the factorial *fac*

One possible solution:

Helper function *fac\_rec* receives parameter *n* and itself as *f*, then calls *f*.

## Example - the factorial *fac*

One possible solution:

Helper function *fac\_rec* receives parameter *n* and itself as *f*, then calls  $f.fac\_rec := \lambda fn. (iszero\ n)\ 1\ (*n\ (f\ f\ (pred\ n)))$

## Example - the factorial *fac*

One possible solution:

Helper function *fac\_rec* receives parameter *n* and itself as *f*, then calls  $f.fac\_rec := \lambda fn. (iszero\ n)\ 1\ (*n\ (f\ f\ (pred\ n)))$

$fac := \lambda n. fac\_rec\ fac\_rec\ n$

Alternatively: recursion with fixed point operator

# Advantages and disadvantages

## Advantages

- very simple definition
- is both a “programming language” and a mathematical construct about which mathematical statements can be made
- no side effects

# Advantages and disadvantages

## Advantages

- very simple definition
- is both a “programming language” and a mathematical construct about which mathematical statements can be made
- no side effects

## Disadvantages

- no “type safety” - functions for booleans can be applied to numbers, pairs, ...  $\Rightarrow$  mistakes can produce results that are difficult to comprehend
- even “simple” operations (subtraction, equality) need many rewriting operations

# Conclusion

- Very useful for mathematical/theoretical purposes (verification, computability and so on)
- Unsuitable as an actual programming language  
But: usable with typing and syntactic sugar (cf. LISP, Haskell, ML) and elements of Lambda calculus have found their way into imperative/object oriented languages (Python, C#, ...)