

The Lambda Calculus and the Functional Programming Paradigm

The theoretical basis of functional programming is the **lambda calculus**.

The Lambda Calculus

$$f(x, y) = x^2 + y$$

The value of f can be written as

$$\lambda x . \lambda y . x^2 + y$$

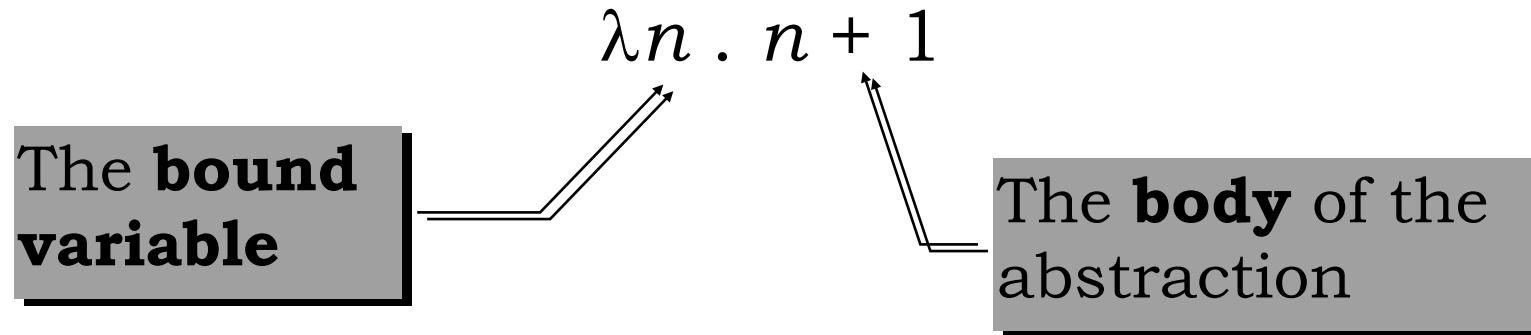
Why we cannot say the value of f is $x^2 + y$?

Syntax of Lambda Calculus

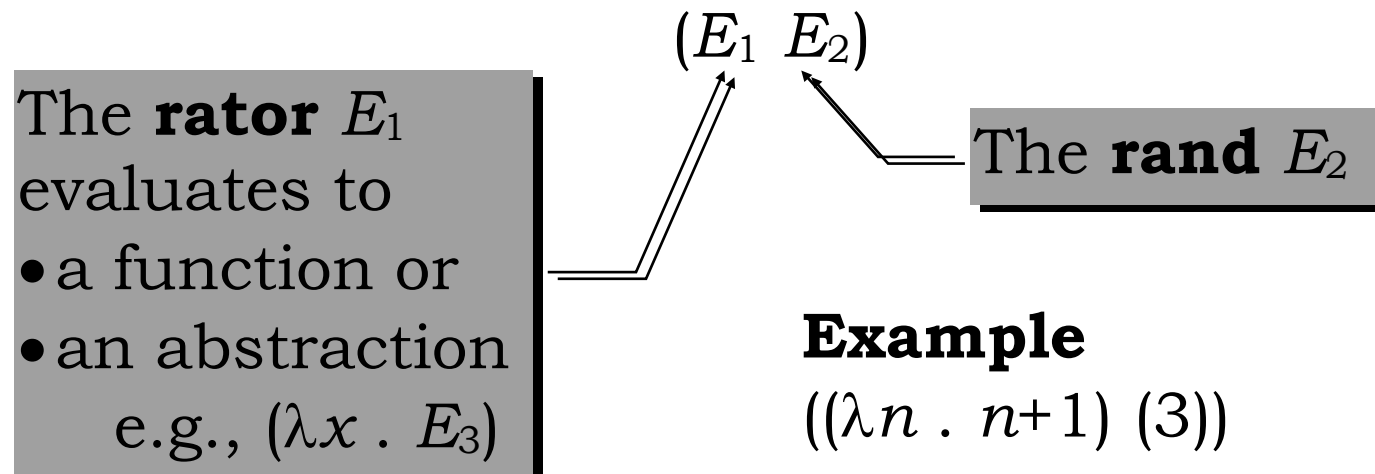
An expression can be

1. A variable
 $x, y, z, time, numberOfPeople, \dots$
2. A predefined constant
— *for impure or applied lambda calculus only*
3. Function applications (combinations)
 $(\text{<expression> } \text{<expression>})$
4. Lambda abstractions (function definitions)
 $(\lambda \text{ <variable> } . \text{ <expression>})$

Lambda abstraction:



Function application:



Some examples

combination	result	remarks
$((\lambda n . n^3) (3))$	27	3^3
$((\lambda x . x) (E))$	E	identity function
$((\lambda n . (\text{add } n \ 1)) (5))$	6	
$((((\lambda f . (\lambda x . (f (f \ x)))) \text{sqr}) \ 3)$	81	$(\text{sqr} (\text{sqr} \ 3))$

Many parentheses!!

This is a **polymorphic operation**:
allow many argument types

Syntactic conventions:

1. Uppercase letters and identifiers are used to denote lambda expressions.
2. $E_1 E_2 E_3$ means $((E_1 E_2) E_3)$.
3. $\lambda x . E_1 E_2 E_3$ means $(\lambda x . (E_1 E_2 E_3))$.
4. $\lambda x y z . E$ means $(\lambda x . (\lambda y . (\lambda z . E)))$.
5. Functions can be named

define Twice = $\lambda f . \lambda x . f (f x)$

then

$(\text{Twice } (\lambda n . (\text{add } n \ 1)) \ 5) = 7$

$((\lambda f . \lambda x . f (f x)) (\lambda n . (\text{add } n \ 1)) \ 5) = 7$

Example

$$(\lambda n . \lambda f . \lambda x . f (n f x)) (\lambda g . \lambda y . g y)$$

What does this mean?

$(\lambda n . \lambda f . \lambda x . f (n f x))$	$(\lambda g . \lambda y . g y)$
$(\lambda n . (\lambda f . \lambda x . f (n f x)))$	$(\lambda g . (\lambda y . g y))$
$(\lambda n . (\lambda f . (\lambda x . f (n f x))))$	$(\lambda g . (\lambda y . (g y)))$
$(\lambda n . (\lambda f . (\lambda x . (f (n f x)))))$	$(\lambda g . (\lambda y . (g y)))$

Curried Functions

$\lambda x . E$

One Parameter Only?

Two ways to represent functions with more than one parameter.

Example: $\text{sum}(a, b) = a + b$
 $(\lambda a . (\lambda b . (\text{add } a \ b)))$

- $\text{sum} : N \times N \rightarrow N$

Note: sum can be seen as $\text{sum}(\langle a, b \rangle) = a + b$.

- $(\lambda a . (\lambda b . (\text{add } a \ b))) : N \rightarrow (N \rightarrow N)$

Note: $((\lambda a . (\lambda b . (\text{add } a \ b))) \ 1) = (\lambda b . (\text{add } 1 \ b))$

Computationally these two functions are the same, but their signatures are different.

Currying and Uncurrying

define Curry = $\lambda f . \lambda x . \lambda y . f \langle x, y \rangle$

define Uncurry = $\lambda f . \lambda p . f (\text{head } p) (\text{tail } p)$

Example

Curry sum

= $(\lambda f . (\lambda x . \lambda y . f \langle x, y \rangle)) \text{ sum}$

= $(\lambda x . \lambda y . \text{sum } \langle x, y \rangle)$

= $(\lambda x . (\lambda y . (\text{add } x y)))$

Uncurry $(\lambda x . (\lambda y . (\text{add } x y)))$

= $(\lambda f . (\lambda p . f (\text{head } p) (\text{tail } p))) (\lambda x . (\lambda y . (\text{add } x y)))$

= $(\lambda p . \text{add } (\text{head } p) (\text{tail } p))$

= $(\lambda p . \text{add } p) = \text{sum}$

‘Partial application’ via currying

define Twice = $\lambda f . \lambda x . f (f x)$

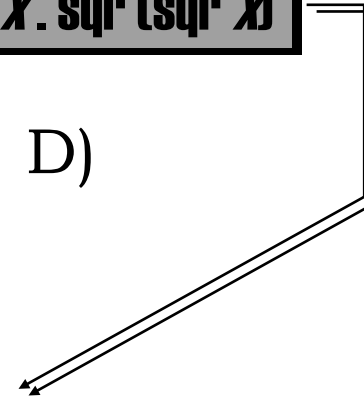
The signature of Twice:

$\lambda x . \text{sqr} (\text{sqr } x)$

Twice : $(D \rightarrow D) \rightarrow (D \rightarrow D)$

Hence

$(\text{Twice } \text{sqr}) : \mathbb{N} \rightarrow \mathbb{N}$

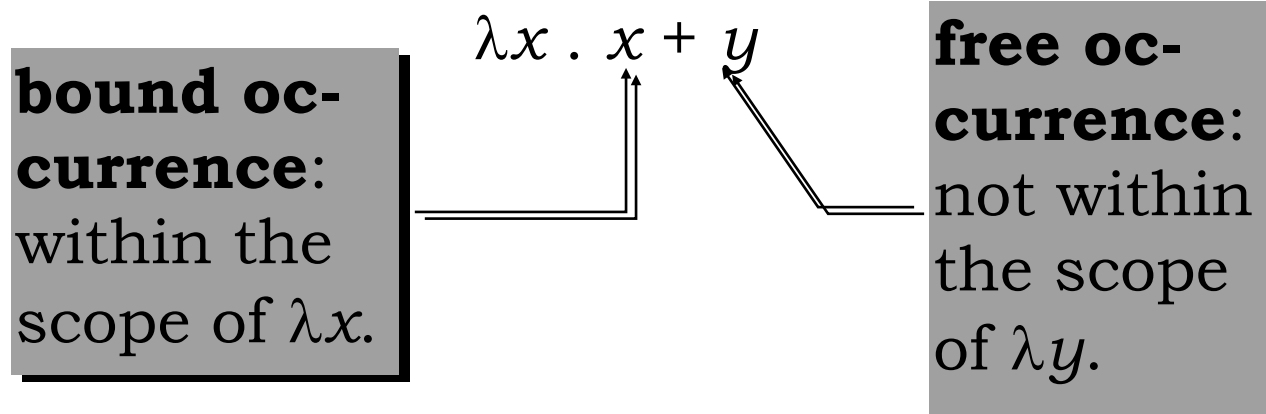


Semantics of Lambda Expressions

Meaning of a lambda expression: results after all its combinations are carried out.

$$((\lambda n . n^3) (3)) = 27$$

Free and bound occurrences of a variable



Substitution

$$E[v \rightarrow E_1]$$

Replace each free occurrence of v in E by E_1 .

$(\lambda x . (\text{mul } y \ x))[y \rightarrow 3]$ $(\lambda x . (\text{mul } 3 \ x))$ **valid!**

A substitution is **valid** (or **safe**) if no free variable in E_1 becomes bound as a result.

$(\lambda x . (\text{mul } y \ x))[y \rightarrow x]$ $(\lambda x . (\text{mul } x \ x))$ **unsafe!**
*This is called a **variable capture** or a **name clash**.*

Definition

$FV(E)$: set of free variables in E .

1. $FV(c) = \emptyset$ (c is a constant)
2. $FV(x) = \{x\}$ (x is a variable)
3. $FV(E_1 E_2) = FV(E_1) \cup FV(E_2)$
4. $FV(\lambda x . E) = FV(E) \setminus \{x\}$

E is **closed** iff $FV(E) = \emptyset$.

Formal definition of substitution

- a) $v[v \rightarrow E_1] = E_1$ for any variable v
- b) $x[v \rightarrow E_1] = x$ for any variable $x \neq v$
- c) $c[v \rightarrow E_1] = c$ for any constant c
- d) $(E_1 E_2)[v \rightarrow E_3] = ((E_1[v \rightarrow E_3]) (E_2[v \rightarrow E_3]))$
- e) $(\lambda v . E)[v \rightarrow E_1] = (\lambda v . E)$
- f) $(\lambda x . E)[v \rightarrow E_1] = \lambda x . (E[v \rightarrow E_1])$
if $x \neq v$ and $x \notin FV(E_1)$
- g) $(\lambda x . E)[v \rightarrow E_1] = \lambda z . (E[x \rightarrow z][v \rightarrow E_1])$
if $x \neq v$ and $x \in FV(E_1)$ and
 $z \neq v$ and $z \notin FV(E E_1)$

Lambda Reduction


Evaluation of a lambda expression:
reduce the expression until no more reduction rules apply.

Main reduction rule:

β -reduction (function application)

$$(\lambda v . E) E_1 \Rightarrow_{\beta} E[v \rightarrow E_1]$$

L.H.S.: a **β -redex** (reduction expression)



Name changing rule:

α -reduction (safe substitution for free variables)

$$\lambda v . E \Rightarrow_{\alpha} \lambda w . E[v \rightarrow w]$$

provided that w does not occur in E at all.

Evaluation is a sequence of β -reductions and α -reductions. **Goal of evaluation** is a lambda expression that contains no β -redexes.

Notation:

$$\mathbf{E} \Rightarrow \mathbf{F} : E \Rightarrow_{\alpha} F \text{ or } E \Rightarrow_{\beta} F$$

$\mathbf{E} \Rightarrow^* \mathbf{F}$: the reflexive, transitive closure of ' \Rightarrow '

Reversed β -reduction

β -abstraction

$$E[v \rightarrow E_1] \Rightarrow_{\beta} (\lambda v. E) E_1.$$

β -conversion

$$E \Leftrightarrow_{\beta} F \text{ if } E \Rightarrow_{\beta} F \text{ or } F \Rightarrow_{\beta} E.$$

E and F are **equivalent** (or **equal**) if $E \Leftrightarrow^* F$.

Equivalence of functions

η -reduction

$$\lambda v . (E \ v) \Rightarrow_{\eta} E$$

provided that E denotes a function and v has no free occurrence in E .

Example: $\lambda x . (\text{sqr } x) \Rightarrow_{\eta} \text{sqr}$

For applied lambda calculus (with constants)

δ -reduction

all rules that involve predefined constants

Examples: $(\text{add } 3 \ 5) \Rightarrow_{\delta} 8$
 $(\text{not true}) \Rightarrow_{\delta} \text{false}$

$$\begin{aligned}
& \text{Twice } (\lambda n . (\text{add } n \ 1)) \ 5 \\
= & (\lambda f . \lambda x . (f \ (f \ x))) \ (\lambda n . (\text{add } n \ 1)) \ 5 \\
\Rightarrow_{\beta} & (\lambda x . ((\lambda n . (\text{add } n \ 1)) \ ((\lambda n . (\text{add } n \ 1)) \ x))) \ 5 \\
\Rightarrow_{\beta} & (\lambda n . (\text{add } n \ 1)) \ ((\lambda n . (\text{add } n \ 1)) \ 5) \\
\Rightarrow_{\beta} & (\text{add } ((\lambda n . (\text{add } n \ 1)) \ 5) \ 1) \\
\Rightarrow_{\beta} & (\text{add } (\text{add } 5 \ 1) \ 1) \\
\Rightarrow_{\delta} & 7
\end{aligned}$$

Normal Form

A lambda expression is in **normal form** if it contains no β -redexes (and no δ -redexes in an applied lambda calculus), so that it cannot be further reduced using the β -rule or the δ -rule.

An expression in normal form has no more function applications (combinations) to evaluate.

Not all lambda expressions can be reduced to normal form.

$$(\lambda x . x x) (\lambda x . x x) \Rightarrow_{\beta} (\lambda x . x x) (\lambda x . x x)$$

Reduction Strategies

Normal order reduction:

Always reduces the leftmost outermost β -redex (or δ -redex) first.

For any lambda expression of the form

$$E = ((\lambda x . B) A)$$

A β -redex E is outside any β -redex that occurs in B or A . A β -redex is outermost if there is no β -redex outside it.

Applicative order reduction:

Always reduces the leftmost innermost β -redex (or δ -redex) first.

For any lambda expression of the form

$$E = ((\lambda x . B) A)$$

Any β -redex that occurs in B or A is inside the β -redex E . A β -redex is innermost if there is no β -redex inside it.

Uniqueness of Normal Form

Church-Rosser Theorem I

For any lambda expressions E , F and G , if $E \Rightarrow^* F$ and $E \Rightarrow^* G$, then there is a lambda expression Z such that $F \Rightarrow^* Z$ and $G \Rightarrow^* Z$.

Corollary

For any lambda expressions E , M and N , if $E \Rightarrow^* M$ and $E \Rightarrow^* N$ where M and N are in normal form, then M and N are variants of each other (equivalent with respect to a α -reduction).

Completeness of Reduction Strategy

Church-Rosser Theorem II

For any lambda expressions E and N , if $E \Rightarrow^* N$ where N is in normal form, there is a normal order reduction from E to N .

A normal order reduction may

1. reach a unique normal form (up to an α -conversion);
2. never terminate.

Constants

How to represent constants in pure lambda calculus? (Hence δ -reductions are not really needed)

Example: Natural numbers

define 0 = $\lambda f . \lambda x . x$

define 1 = $\lambda f . \lambda x . f x$

define 2 = $\lambda f . \lambda x . f (f x)$

define 3 = $\lambda f . \lambda x . f (f (f x))$

...

define add = $\lambda m . \lambda n . \lambda f . \lambda x . m f (n f x)$

Functional Programming

```
fun factorial (n) =  
  if n > 0  
  then n * factorial (n - 1)  
  else 1
```

```
factorial =  
  λn . if n > 0  
       then n * factorial (n - 1)  
       else 1
```

Functional Programming

- A program is a function composed using simpler functions.
- As a result, there is no variable.
- As loops can be defined using recursively — recursive functions replace loops.