

## Chapter 5: The Untyped Lambda Calculus

What is lambda calculus for?

Basics: syntax and operational semantics

Programming in the Lambda Calculus

Formalities (formal definitions)



# What is Lambda calculus for?

- A **core calculus** (used by Landin) for
  - capturing the language's essential mechanisms,
  - with a collection of convenient derived forms whose behavior is understood by translating them into the core
- A **formal system** invented in the 1920s by Alonzo Church (1936, 1941), in which all **computation** is reduced to the basic operations of function definition and application.



# Basics



# Syntax

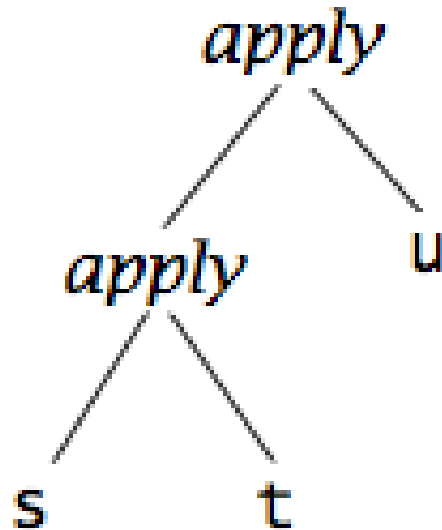
- The **lambda-calculus** (or  $\lambda$ -calculus) embodies this kind of function definition and application in the purest possible form.

$t ::=$	<i>terms:</i>
$x$	<i>variable</i>
$\lambda x. t$	<i>abstraction</i>
$t t$	<i>application</i>



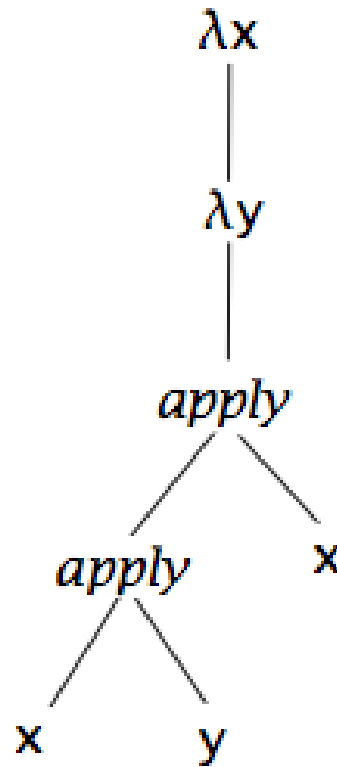
# Abstract Syntax Trees

- (s t) u (or simply written as s t u)



# Abstract Syntax Trees

- $\lambda x. (\lambda y. ((x y) x))$   
(or simply written as  $\lambda x. \lambda y. x y x$ )



# Scope

- An occurrence of the variable  $x$  is said to be **bound** when it occurs in the body  $t$  of an abstraction  $\lambda x.t$ .
  - $\lambda x$  is a **binder** whose **scope** is  $t$ . A binder can be **renamed**: e.g.,  $\lambda x.x = \lambda y.y$ .
- An occurrence of  $x$  is **free** if it appears in a position where it is not bound by an enclosing abstraction on  $x$ .
  - **Exercises**: Find free variable occurrences from the following terms:  $x y$ ,  $\lambda x.x$ ,  $\lambda y. x y$ ,  $(\lambda x.x) x$ .



# Operational Semantics

- Beta-reduction: the only computation

$$(\lambda x. t_{12}) t_2 \rightarrow [x \mapsto t_2]t_{12},$$

“the term obtained by replacing all **free** occurrences of  $x$  in  $t_{12}$  by  $t_2$ ”

A term of the form  $(\lambda x.t_{12}) t_2$  is called a **redex**.

- Examples:

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

$$(\lambda x. x (\lambda x.x)) (u r) \rightarrow u r (\lambda x.x)$$





# Evaluation Strategies

- Full beta-reduction
  - Any redex may be reduced at any time.
- Example:
  - Let  $\text{id} = \lambda x.x$ . We can apply beta reduction to any of the following underlined redexes:

$$\begin{array}{c} \underline{\text{id} (\text{id} (\lambda z. \text{id} z))} \\ \text{id} (\underline{\text{id} (\lambda z. \text{id} z)}) \\ \text{id} (\text{id} (\lambda z. \underline{\text{id} z})) \end{array}$$

Note: lambda calculus is confluent under full beta-reduction. Ref. Church-Rosser property.



# Evaluation Strategies

- The normal order strategy
  - The leftmost, outmost redex is always reduced first.

$$\begin{aligned}
 & \text{id (id (\lambda z. id z))} \\
 \rightarrow & \text{id (\lambda z. id z)} \\
 \rightarrow & \lambda z. \text{id z} \\
 \rightarrow & \lambda z. z \\
 \nrightarrow &
 \end{aligned}$$


# Evaluation Strategies

- The call-by-name strategy
  - A more restrictive normal order strategy, allowing no reduction inside abstraction.

$$\begin{aligned} & \underline{\text{id (id (\lambda z. id z))}} \\ \rightarrow & \underline{\text{id (\lambda z. id z)}} \\ \rightarrow & \lambda z. id z \\ \not\rightarrow & \end{aligned}$$


# Evaluation Strategies

- The call-by-value strategy
  - only outermost redexes are reduced and where a redex is reduced only when its right-hand side has already been reduced to a value
  - Value: a term that cannot be reduced any more.

$\text{id } (\text{id } (\lambda z. \text{id } z))$   
 $\rightarrow \text{id } (\lambda z. \text{id } z)$   
 $\rightarrow \lambda z. \text{id } z$   
 $\nrightarrow$



# Call-by-name vs Call-by-value

- Call-by-name
  - $(\lambda x. y) ((\lambda x. x) z) = y$
- Call-by-value
  - $(\lambda x. y) ((\lambda x. x) z) = (\lambda x. y) z = y$



# Programming in the Lambda Calculus

Multiple Arguments

Church Booleans

Pairs

Church Numerals

Recursion

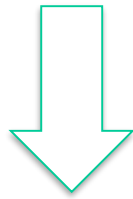


# Multiple Arguments

$$f(x, y) = s$$

currying 

$$(f\ x)\ y = s$$



$$f = \lambda x. (\lambda y. s)$$



# Church Booleans

- Boolean values can be encoded as:

$$\text{tru} = \lambda t. \lambda f. t$$

$$\text{fls} = \lambda t. \lambda f. f$$

- Boolean conditional and operators can be encoded as:

$$\text{test} = \lambda l. \lambda m. \lambda n. l m n$$

$$\text{and} = \lambda b. \lambda c. b c \text{ fls}$$




# Church Booleans

- An Example

$$\begin{aligned}
 & \text{test tru v w} \\
 = & \quad \underline{(\lambda l. \lambda m. \lambda n. l m n)} \text{ tru v w} \\
 \rightarrow & \quad \underline{(\lambda m. \lambda n. \text{tru m n})} \text{ v w} \\
 \rightarrow & \quad \underline{(\lambda n. \text{tru v n})} \text{ w} \\
 \rightarrow & \quad \text{tru v w} \\
 = & \quad \underline{(\lambda t. \lambda f. t)} \text{ v w} \\
 \rightarrow & \quad \underline{(\lambda f. v)} \text{ w} \\
 \rightarrow & \quad \text{v}
 \end{aligned}$$



# Church Numerals

- Encoding Church numerals:

$$c_0 = \lambda s. \lambda z. z;$$

$$c_1 = \lambda s. \lambda z. s z;$$

$$c_2 = \lambda s. \lambda z. s (s z);$$

$$c_3 = \lambda s. \lambda z. s (s (s z));$$

etc.

- Defining functions on Church numerals:

$$\text{succ} = \lambda n. \lambda s. \lambda z. s (n s z);$$

$$\text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z);$$

$$\text{times} = \lambda m. \lambda n. m (\text{plus } n) c_0;$$



# Church Numerals

- Can you define minus?
- Suppose we have pred, can you define minus?
  - $\lambda m. \lambda n. n \text{ pred } m$
- Can you define pred?
  - $\lambda n. \lambda s. \lambda z. n (\lambda g. \lambda h. h (g s)) (\lambda u. z) (\lambda u. u)$
  - $(\lambda u. z)$  -- a wrapped zero
  - $(\lambda u. u)$  – the last application to be skipped
  - $(\lambda g. \lambda h. h (g s))$  -- apply h if it is the last application, otherwise apply g
  - Try  $n = 0, 1, 2$  to see the effect



# Pairs

- Encoding

```
pair = λf.λs.λb. b f s;
fst  = λp. p tru;
snd  = λp. p fls;
```

- An Example

```
fst (pair v w)
= fst ((λf. λs. λb. b f s) v w)
→ fst ((λs. λb. b v s) w)
→ fst (λb. b v w)
= (λp. p tru) (λb. b v w)
→ (λb. b v w) tru
→ tru v w
→* v
```



# Recursion

- Terms with no normal form are said to **diverge**.

$$\text{omega} = (\lambda x. x x) (\lambda x. x x);$$

- Fixed-point combinator

$$\text{fix} = \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y));$$

Note:  $\text{fix } f = f (\lambda y. (\text{fix } f) y)$



# Recursion

- Basic Idea:

A recursive definition:  $h = \langle \text{body containing } h \rangle$



$g = \lambda f . \langle \text{body containing } f \rangle$

$h = \text{fix } g$



# Recursion

- Example:

$fac = \lambda n. \text{if eq } n \text{ } c_0$   
      $\text{then } c_1$   
      $\text{else times } n \text{ (fac (pred } n))$



$g = \lambda f . \lambda n. \text{if eq } n \text{ } c_0$   
      $\text{then } c_1$   
      $\text{else times } n \text{ (f (pred } n))$

$fac = \text{fix } g$

**Exercise:** Check that  $fac \text{ } c_3 \rightarrow c_6$ .



# Y Combinator

$$Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

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

- $Y f = f (Y f)$
- Why fix is used instead of Y?





# Answer

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

$$Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

- Assuming call-by-value
  - $(x x)$  is not a value
  - while  $(\lambda y. x x y)$  is
  - $Y$  will diverge for any  $f$



# Formalities (Formal Definitions)

Syntax (free variables)

Substitution

Operational Semantics



# Syntax

- **Definition** [Terms]: Let  $V$  be a countable set of variable names. The set of terms is the smallest set  $T$  such that
  1.  $x \in T$  for every  $x \in V$ ;
  2. if  $t_1 \in T$  and  $x \in V$ , then  $\lambda x.t_1 \in T$ ;
  3. If  $t_1 \in T$  and  $t_2 \in T$ , then  $t_1 t_2 \in T$ .

- Free Variables

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

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

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



# Substitution

$$\begin{aligned} [x \mapsto s]x &= s \\ [x \mapsto s]y &= y && \text{if } y \neq x \\ [x \mapsto s](\lambda y. t_1) &= \lambda y. [x \mapsto s]t_1 && \text{if } y \neq x \text{ and } y \notin FV(s) \\ [x \mapsto s](t_1 t_2) &= [x \mapsto s]t_1 [x \mapsto s]t_2 \end{aligned}$$

Alpha-conversion: Terms that differ only in the names of bound variables are interchangeable in all contexts.

Example:

$$\begin{aligned} & [x \rightarrow y z] (\lambda y. x y) \\ = & [x \rightarrow y z] (\lambda w. x w) \\ = & \lambda w. y z w \end{aligned}$$



# Operational Semantics

## Syntax

$t ::=$

$x$

$\lambda x. t$

$t t$

*terms:*  
*variable*  
*abstraction*  
*application*

$v ::=$

$\lambda x. t$

*values:*  
*abstraction value*

## Evaluation

$t \rightarrow t'$

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2}$$

(E-APP1)

$$\frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2}$$

(E-APP2)

$$(\lambda x. t_{12}) v_2 \rightarrow [x \mapsto v_2] t_{12} \quad \text{(E-APPABS)}$$



# Summary



- What is lambda calculus for?
  - A core calculus for capturing language essential mechanisms
  - Simple but powerful
- Syntax
  - Function definition + function application
  - Binder, scope, free variables
- Operational semantics
  - Substitution
  - Evaluation strategies: normal order, call-by-name, call-by-value



# Homework

- Understand Chapter 5.
- Do exercise 5.3.6 in Chapter 5.

5.3.6 EXERCISE [★★]: Adapt these rules to describe the other three strategies for evaluation—full beta-reduction, normal-order, and lazy evaluation. □

