# 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

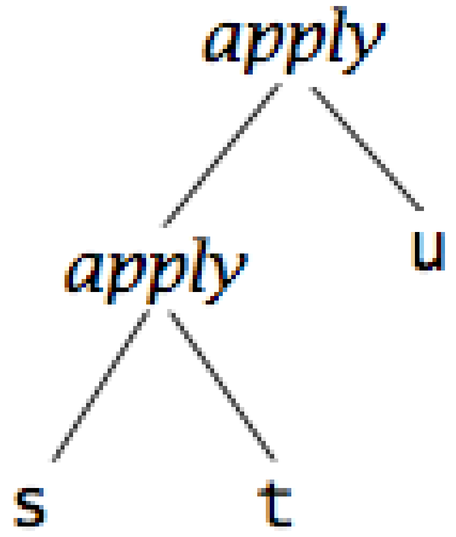
# Syntax

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

| | | | |
|---|---|---|---|
| t | ::= | | *terms:* |
| | | x | *variable* |
| | | λx.t | *abstraction* |
| | | t t | *application* |

# Abstract Syntax Trees

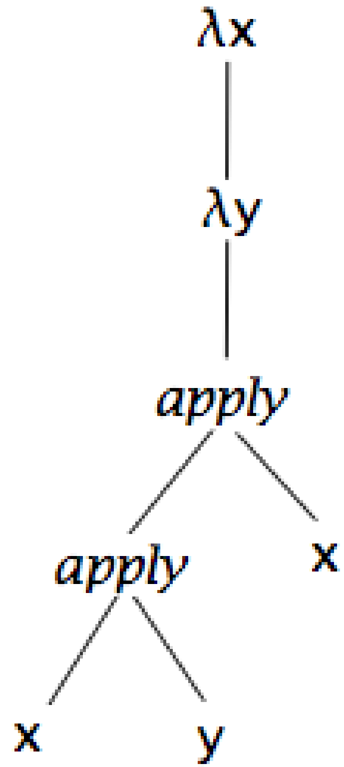
- (s t) u   (or simply written as s t u)



# Abstract Syntax Trees

- λx. (λy. ((x y) x))

  (or simply written as λx. λy. 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 λx.t.
  - λx is a binder whose scope is t. A binder can be renamed: e.g., λx.x = λy.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, λx.x, λy. x y, (λx.x) x.

# Operational Semantics

- Beta-reduction: the only computation

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

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

A term of the form (λx.t12) t2 is called a redex.

- Examples:

(λx.x) y → y

(λx. x (λx.x)) (u r) → u r (λx.x)

# Evaluation Strategies

- Full beta-reduction
  - Any redex may be reduced at any time.

- Example:
  - Let id = λx.x. We can apply beta reduction to any of the following underlined redexes:

$$\underline{\mathtt{id}\ (\mathtt{id}\ (\lambda z.\ \mathtt{id}\ z))}$$

$$\mathtt{id}\ (\underline{(\mathtt{id}\ (\lambda z.\ \mathtt{id}\ z))})$$

$$\mathtt{id}\ (\mathtt{id}\ (\lambda z.\ \underline{\mathtt{id}\ z}))$$

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.

$$\begin{array}{ll} & \underline{\mathtt{id}\,(\mathtt{id}\,(\lambda z.\,\mathtt{id}\,z))} \\ \longrightarrow & \underline{\mathtt{id}\,(\lambda z.\,\mathtt{id}\,z)} \\ \longrightarrow & \lambda z.\,\underline{\mathtt{id}\,z} \\ \longrightarrow & \lambda z.z \\ \not\longrightarrow & \end{array}$$

# Evaluation Strategies

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

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

# 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.

$$
\begin{array}{ll}
 & \mathtt{id}\ \underline{(\mathtt{id}\ (\lambda z.\ \mathtt{id}\ z))} \\
\longrightarrow & \underline{\mathtt{id}\ (\lambda z.\ \mathtt{id}\ z)} \\
\longrightarrow & \lambda z.\ \mathtt{id}\ z \\
\not\longrightarrow &
\end{array}
$$

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

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

# 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:

  tru = λt. λf. t

  fls = λt. λf. f

- Boolean conditional and operators can be encoded as:

  test = λl. λm. λn. l m n

  and = λb. λc. b c fls

# Church Booleans

- An Example

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

# Church Booleans

- Can you define *or*?

- $or = \lambda a. \lambda b. a\ tru\ b$

# 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:

succ = λn. λs. λz. s (n s z);

plus = λm. λn. λs. λz. m s (n s z);

times = λm. λn. m (plus n) c0;

# Church Numerals

- Can you define minus?

- Suppose we have pred, can you define minus?
  - $\lambda m. \lambda n. n\ pred\ m$

- Can you define pred?
  - $\lambda n. \lambda s. \lambda z. n\ \left(\lambda g. \lambda h. h\ (g\ s)\right)\ (\lambda u. z)\ (\lambda u. u)$
  - $(\lambda u. z)$ -- a wrapped zero
  - $(\lambda u. u)$ – the last application to be skipped
  - $\left(\lambda g. \lambda h. h\ (g\ s)\right)$ -- 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

$$
\begin{array}{ll}
 & \mathtt{fst\ (pair\ v\ w)} \\
= & \mathtt{fst\ (}\underline{\mathtt{(\lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s)\ v}}\mathtt{\ w)} \\
\longrightarrow & \mathtt{fst\ (}\underline{\mathtt{(\lambda s.\ \lambda b.\ b\ v\ s)\ w}}\mathtt{)} \\
\longrightarrow & \mathtt{fst\ (\lambda b.\ b\ v\ w)} \\
= & \underline{\mathtt{(\lambda p.\ p\ tru)\ (\lambda b.\ b\ v\ w)}} \\
\longrightarrow & \underline{\mathtt{(\lambda b.\ b\ v\ w)\ tru}} \\
\longrightarrow & \mathtt{tru\ v\ w} \\
\longrightarrow^{*} & \mathtt{v}
\end{array}
$$

# Recursion

- Terms with no normal form are said to diverge.

  omega = (λx. x x) (λx. x x);

- Fixed-point combinator

  fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y));

  Note: fix f = f (λy. (fix f) y)

# Recursion

- Basic Idea:

A recursive definition: h = <body containing h>

g = λf . <body containing f>

h = fix g

# Recursion

- Example:

fac = λn. if eq n c0

then c1

else times n (fac (pred n))

g = λf . λn. if eq n c0

then c1

else times n (f (pred n))

fac = fix g

**Exercise**: Check that fac c3 → c6.

# Y Combinator

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

```
fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))
```

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

# Answer

```
fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))
```

```
Y = λf. (λx. f (x x)) (λx. f (x x))
```

- Assuming call-by-value
  - `(x x)` is not a value
  - while `(λy. 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 t1 $\in$ 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

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

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

Example:

$[x \rightarrow y\ z]\ (\lambda y.\ x\ y)$

$= [x \rightarrow y\ z]\ (\lambda w.\ x\ w)$

$= \lambda w.\ y\ z\ w$

# Operational Semantics

*Syntax*

| | | | |
|---|---|---|---|
| $t$ | ::= | | *terms:* |
| | | $x$ | *variable* |
| | | $\lambda x.t$ | *abstraction* |
| | | $t\ t$ | *application* |
| $v$ | ::= | | *values:* |
| | | $\lambda x.t$ | *abstraction value* |

*Evaluation* $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

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

# 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. □