编程语言的设计原理

Design Principles of Programming Languages

Zhenjiang Hu, Haiyan Zhao,
胡振江 赵海燕
Peking University, Spring, 2022

Recap

- Core messages in the previous lecture
  - (Untyped) programming languages are defined by *syntax* and *semantics*
  - Syntax is often specified by grammars
    - Inductively vs structural induction
  - Semantics can be specified in three ways, and this book chooses *operational semantics* expressed as *evaluation rules*
  - Big step vs small step semantics

Abstract Machines

- An abstract machine consists of:
  – a set of *states*
  – a *transition relation* on states, written $\longrightarrow$
    "$t \longrightarrow t'$" is read as "t evaluates to t' in *one step*".

- A *state* records all the information in the abstract machine at a given moment.
  – e.g., an abstract-machine-style description of a conventional microprocessor would include the program counter, the contents of the registers, the contents of main memory, and the machine code program being executed.

Operational semantics for Booleans

- Syntax of terms and values

```
t ::=                                  terms
        true                           constant true
        false                          constant false
        if t then t else t             conditional

v ::=                                  values
        true                           true value
        false                          false value
```

Evaluation relation for Booleans

- The evaluation relation $t \longrightarrow t'$ is the smallest relation closed under the following rules:

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \qquad \text{(E-IfTrue)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \qquad \text{(E-If)}$$

Evaluation relation for Booleans

- Computation rules

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \quad \text{(E-IfTrue)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \quad \text{(E-IfFalse)}$$

- Congruence rules

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \quad \text{(E-If)}$$

- Computation rules perform *"real" computation* steps
- Congruence rules determine *where computation rules* can be *applied* next

Evaluation relation for Booleans

$\longrightarrow$ is the smallest two-place relation closed under the following rules:

$$((\texttt{if true then } t_2 \texttt{ else } t_3), t_2) \in \longrightarrow$$

$$((\texttt{if false then } t_2 \texttt{ else } t_3), t_3) \in \longrightarrow$$

$$\frac{(t_1, t_1') \in \longrightarrow}{((\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3), (\texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3)) \in \longrightarrow}$$

The notation $t \longrightarrow t'$ is short-hand for $(t, t') \in \longrightarrow$.

If the pair (t, t') is an evaluation relation, then the evaluation statement or judgement $t \longrightarrow t'$ is said to be derivable

Derivation

- "Justification" for a particular pair of terms that are in the evaluation relation in *the form of a tree*.

$$\dfrac{\dfrac{\dfrac{}{\texttt{s} \longrightarrow \texttt{false}}\ \textsc{E-IfTrue}}{\texttt{t} \longrightarrow \texttt{u}}\ \textsc{E-If}}{\texttt{if t then false else false} \longrightarrow \texttt{if u then false else false}}\ \textsc{E-If}$$

  – These trees are called derivation trees (or just derivations).
  – The final statement in a derivation is its conclusion.
  – We say that the derivation is a witness for its conclusion (or a proof of its conclusion) — it records all the reasoning steps that justify the conclusion.

Induction on Derivation

$$\dfrac{\dfrac{\dfrac{}{\texttt{s} \longrightarrow \texttt{false}}\ \textsc{E-IfTrue}}{\texttt{t} \longrightarrow \texttt{u}}\ \textsc{E-If}}{\texttt{if t then false else false} \longrightarrow \texttt{if u then false else false}}\ \textsc{E-If}$$

- Write proofs about evaluation "*by induction on derivation trees.*"
- Given an arbitrary derivation $\mathcal{D}$ with conclusion $t \longrightarrow t'$, we assume the desired result for its *immediate sub-derivation* (if any) and proceed by *a case analysis* of the *final evaluation rule* used in constructing the derivation tree.

Chapter 5:
The Untyped Lambda Calculus

What is lambda calculus for ?

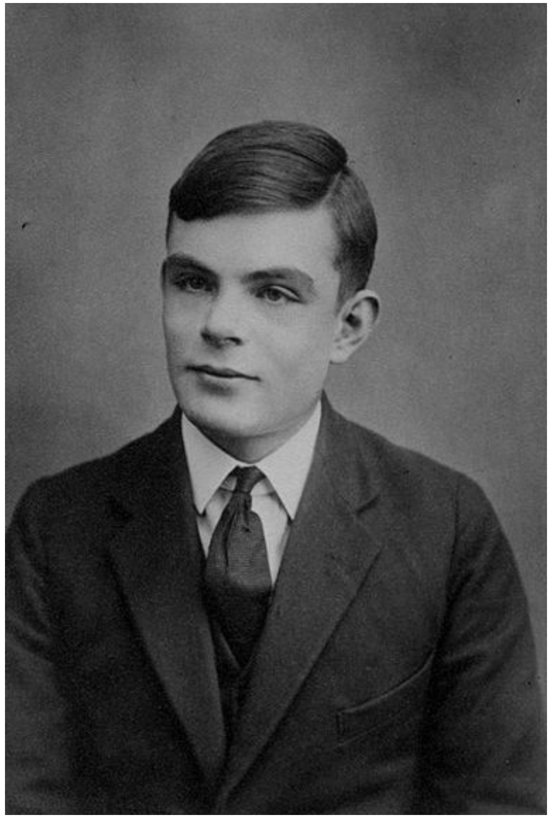
Basics: Syntax and Operational semantics

Programming in the Lambda Calculus

Formalities (formal definitions)

Story of Turing and Church

Alonzo Church
Lambda Calculus



Alan Turing
Turing Machine

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.
  - modeling programming language, as the foundation of many real-world programming language designs (including ML, Haskell, Scheme, Lisp, ...) , and being *central to contemporary computer science*.

Lambda calculus

- A formal system devised by Alonzo Church in the 1930's as a model for computability
  – *all computation* is reduced to the *basic operations of function abstraction and application*.
- A very simple but very powerful language based on pure abstraction
  – Turing complete
  – higher order (functions as data)

Basics

Syntax

Scope

Operational semantics

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*

- Terminology:
  – terms in the pure λ-calculus are often called λ-*terms*
  – terms of the form λx. t are called λ-*abstractions* or just abstractions

Syntactic conventions

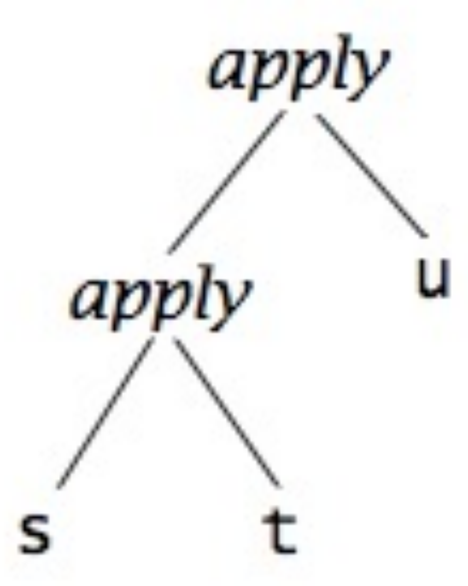
- The λ-calculus provides only one-argument functions, all multi-argument functions must be written in curried style.
- The following *conventions* make the linear forms of terms easier to read and write:
  - Application *associates to the left*

    e.g., *t u v* means *(t u) v*, not *t (u v)*
  - Bodies of λ- abstractions *extend as far to the right as possible*

    e.g., *λx. λy. x y* means *λx. (λy. x y)*, not *λx. (λy. x) y*

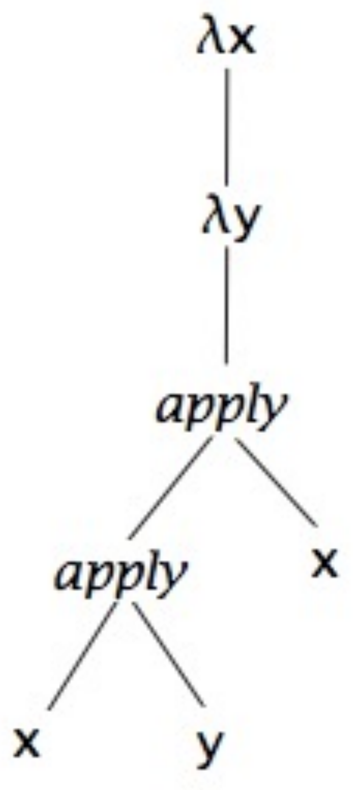
Abstract Syntax Trees

- (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 $\lambda x.t$, i.e.,
  - the λ-abstraction term $\lambda x.t$ binds the variable x, and the scope of this binding is the body t.
  - λx is a *binder* whose *scope* is t.
  - a binder can be *renamed* as necessary
    - so-called: *alpha-renaming*
    - e.g., $\lambda x.\, x = \lambda y.\, y$,

Scope

- An occurrence of *x* is *free* if it appears in a position where it is not bound *by an enclosing abstraction* on *x*.
  - a term with no free variable is said to be *closed*.
  - closed terms are also called *combinators*.
- **Exercises**: Find free variable occurrences from the following terms:
  - x y,
  - λx.x
  - λy. x y
  - (λx.x) x
  - (λx.x) (λy.y x)
  - (λx.x) (λx.x)
  - (λx.(λy.x y)) y

Values

v	::=		*values*
		λx.t	*abstraction value*

Operational Semantics

- *Beta-reduction*: the only computation (substitution)

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

  – the term obtained by *replacing all free occurrences* of x in t_{12} by t_2
  – a term of the form *(λx.t) v* — a *λ-abstraction* applied to a *value* — is called a *redex* (short for "*reducible expression*").

- Examples:

  (λx. x) y → y

  (λx. x (λx .x)) (u r) → u r (λx. x)

Operational Semantics

- If the function $\lambda x.t$ is applied to t_2, we substitute *all free occurrences of* x in t with t_2.
  - If the substitution would bring a free variable of t_2 in an expression *where this variable occurs bound*, we *rename the bound variable* before performing the substitution.

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

- Examples:

  (λx.x) (λx.x) → ?

  (λx.(λy.x y)) y → ?

  (λx.(λy.(x (λx.x y)))) y → ?

Evaluation Strategies

- Full beta-reduction
  - *any redex* may be reduced *at any time*.
- e. g., *id* = *λx.x*
  - we can apply *full beta reduction* to *any* of the following *underlined redexes*:

```
id (id (λz. id z))
id ((id (λz. id z)))
id (id (λz. 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*.
    - try to reduce always the leftmost expression of a series of applications, and continue until *no further reductions* are possible
  - the evaluation relation under this strategy is actually a partial function: each term *t* evaluates in one step to at most one term *t'*

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

Evaluation Strategies

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

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

  - stop before the last and *regard* $\lambda z.\ \mathrm{id}\ z$ as a *normal form*

Evaluation Strategies

- *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.*

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

Evaluation Strategies

- *call-by-value* strategy
  - strict in the sense that *the arguments to functions are always evaluated*, *whether or not they are used* by the body of the function.
  - reflects standard conventions found in most mainstream languages.

Operational Semantics

- Computation rule

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

- Congruence rules

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

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

Lambda Calculus

- Once we have λ-abstraction and application, we can throw away all the other language primitives and still have left a rich and powerful programming language.
- Everything is a function
  - Variables always denote functions
  - Functions always take other functions as parameters
  - The result of a function is always a function

Abstractions over Functions

- Consider the λ-abstraction

$$g = \lambda f.\ f\ (f\ (succ\ 0))$$

  – the parameter variable f is used in the function position in the body of g.
  – terms like g are called higher-order functions.
  – If we apply g to an argument like *plus3*, the "substitution rule" yields a nontrivial computation:

```
g plus3
   =    (λf. f (f (succ 0))) (λx. succ (succ (succ x)))
   i.e. (λx. succ (succ (succ x)))
          ((λx. succ (succ (succ x))) (succ 0))
   i.e. (λx. succ (succ (succ x)))
          (succ (succ (succ (succ 0))))
   i.e. succ (succ (succ (succ (succ (succ (succ 0))))))
```

Programming in the Lambda Calculus

Multiple Arguments

Church Booleans

Pairs

Church Numerals

Recursion

Multiple Arguments

- λ-calculus provides only one-argument functions, all multi-argument functions must be written in curried style.

$$f\ (x, y) = t$$

currying ↓

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

λ-encoding ↓

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

Multiple Arguments

- In general, $\lambda x. \lambda y. t$ is a function that, given a value v for x, yields a function that, given a value u for y, yields t with v in place of x and u in place of y.
  - i.e., $\lambda x. \lambda y. t$ is a *two-argument function*.
- λ-abstraction that does nothing but immediately yields another abstraction — is very common in the λ-calculus.

Church Booleans

- Boolean values can be encoded as:

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

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

	`tru v w`	
$=$	$\underline{(\lambda t.\lambda f.t)\ v}\ w$	by definition
$\longrightarrow$	$\underline{(\lambda f.\ v)\ w}$	reducing the underlined redex
$\longrightarrow$	v	reducing the underlined redex

	`fls v w`	
$=$	$\underline{(\lambda t.\lambda f.f)\ v}\ w$	by definition
$\longrightarrow$	$\underline{(\lambda f.\ f)\ w}$	reducing the underlined redex
$\longrightarrow$	w	reducing the underlined redex

Church Booleans

- Boolean conditional and operators can be encoded as:

  test = $\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n$

	test tru v w	
=	(λl. λm. λn. l m n) tru v w	by definition
⟶	(λm. λn. tru m n) v w	reducing the underlined redex
⟶	(λn. tru v n) w	reducing the underlined redex
⟶	tru v w	reducing the underlined redex
=	(λt.λf.t) v w	by definition
⟶	(λf. v) w	reducing the underlined redex
⟶	v	reducing the underlined redex

Church Booleans

- How to define *not*?
  - a function that, given a boolean value v, returns fls if v is tru and tru if v is fls.

$$\text{not} = \lambda b.\ b\ \text{fls}\ \text{tru}$$

Church Booleans

- Boolean conditional
  - *and* is a function that, given two boolean values v and w, returns w if v is tru and fls if v is fls.
  - thus and v w yields tru if both v and w are tru, and fls if either v or w is fls.

- *and* operators can be encoded as:

$$and = \lambda b.\ \lambda c.\ b\ c\ fls$$

Church Booleans

- How to define *or* ?

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

Church Numerals

- Encoding Church numerals
  – Basic idea: represent the number n by a function that "repeats *some action* n *times*."

$$c_0 = \lambda s.\ \lambda z.\ z$$
$$c_1 = \lambda s.\ \lambda z.\ s\ z$$
$$c_2 = \lambda s.\ \lambda z.\ s\ (s\ z)$$
$$c_3 = \lambda s.\ \lambda z.\ s\ (s\ (s\ z))$$

  – each number n is represented by *a term* c_n taking two arguments, s and z (for "successor" and "zero"), and applies s, n times, to z.

Functions on Church Numerals

- Successor

$$suc \;=\; \lambda n.\lambda s.\lambda z.s\;(n\;s\;z);$$

- addition

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

- Multiplication

$$times \;=\; \lambda m.\lambda n.m\;(plus\;n)\;c0;$$

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

$$\texttt{pair} = \lambda \texttt{f}.\lambda \texttt{s}.\lambda \texttt{b}.\ \texttt{b f s}$$
$$\texttt{fst} = \lambda \texttt{p}.\ \texttt{p tru}$$
$$\texttt{snd} = \lambda \texttt{p}.\ \texttt{p fls}$$

- Example

$$
\begin{array}{lll}
 & \texttt{fst (pair v w)} & \\
= & \texttt{fst (}\underline{\texttt{(}\lambda\texttt{f. }\lambda\texttt{s. }\lambda\texttt{b. b f s) v}}\texttt{ w)} & \text{by definition} \\
\longrightarrow & \texttt{fst (}\underline{\texttt{(}\lambda\texttt{s. }\lambda\texttt{b. b v s) w}}\texttt{)} & \text{reducing} \\
\longrightarrow & \texttt{fst (}\lambda\texttt{b. b v w)} & \text{reducing} \\
= & \underline{\texttt{(}\lambda\texttt{p. p tru) (}\lambda\texttt{b. b v w)}} & \text{by definition} \\
\longrightarrow & \underline{\texttt{(}\lambda\texttt{b. b v w) tru}} & \text{reducing} \\
\longrightarrow & \texttt{tru v w} & \text{reducing} \\
\longrightarrow^{*} & \texttt{v} & \text{as before.}
\end{array}
$$

Recursion

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

- Note that omega evaluates *in one step* to *itself* !
  – evaluation of omega never reaches a normal form: it diverges.

- Terms with no normal form are said to diverge.

- Divergent computation does not seem very useful in itself. However, there are variants of omega that are very useful...

Recursion

- Fixed-point combinator

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

Note that

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

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

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

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

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

Y Combinator

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

$$
\begin{aligned}
Y &= \\
&\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))} \\
&\longrightarrow \\
&f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}) \\
&\longrightarrow \\
&f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))})) \\
&\longrightarrow \\
&f\ (f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}))) \\
&\longrightarrow \\
&\ldots
\end{aligned}
$$

Answer

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


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

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

Formalities (Formal Definitions)

Syntax (free variables)

Substitution

Operational Semantics

Syntax

- **Definition** [Terms]:

  Let $\mathcal{V}$ be a countable set of variable names.

  The set of terms is the smallest set $\mathcal{T}$ such that

  1. $x \in \mathcal{T}$ for every $x \in \mathcal{V}$;
  2. if $t_1 \in \mathcal{T}$ and $x \in \mathcal{V}$, then $\lambda x.t_1 \in \mathcal{T}$;
  3. if $t_1 \in \mathcal{T}$ and $t_2 \in \mathcal{T}$, then $t_1\ t_2 \in \mathcal{T}$.

- 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:

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

Operational Semantics

Syntax

$\texttt{t}$	::=		*terms:*
		$\texttt{x}$	*variable*
		$\lambda\texttt{x.t}$	*abstraction*
		$\texttt{t t}$	*application*
$\texttt{v}$	::=		*values:*
		$\lambda\texttt{x.t}$	*abstraction value*

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

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{t}_1\ \texttt{t}_2 \longrightarrow \texttt{t}_1'\ \texttt{t}_2} \quad \text{(E-App1)}$$

$$\frac{\texttt{t}_2 \longrightarrow \texttt{t}_2'}{\texttt{v}_1\ \texttt{t}_2 \longrightarrow \texttt{v}_1\ \texttt{t}_2'} \quad \text{(E-App2)}$$

$$(\lambda\texttt{x.t}_{12})\ \texttt{v}_2 \longrightarrow [\texttt{x} \mapsto \texttt{v}_2]\texttt{t}_{12} \quad \text{(E-AppAbs)}$$

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

- Read through and understand Chapter 5.
- Do exercise 5.2.7， 5.3.6 in Chapter 5.

5.2.7 EXERCISE [★★]: Write a function `equal` that tests two numbers for equality and returns a Church boolean. For example,

```
  equal c3 c3;
▸ (λt. λf. t)

  equal c3 c2;
▸ (λt. λf. f)
```
□

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