# 18-lambda

November 6, 2026

## 1 CMSC 330: Organization of Programming Languages

### 1.1 Lambda Calculus

---

### 1.2 Previously in CMSC330

- **Operational semantics**: rules for evaluating a small language
- **Type checking** : rules for the judgment $G \vdash e : t$

Each language we studied had integers, booleans, `if`, `let`, records, … built in.

How small can a language get and still compute **anything**?

Today: a language with **only functions** — the **lambda calculus**.

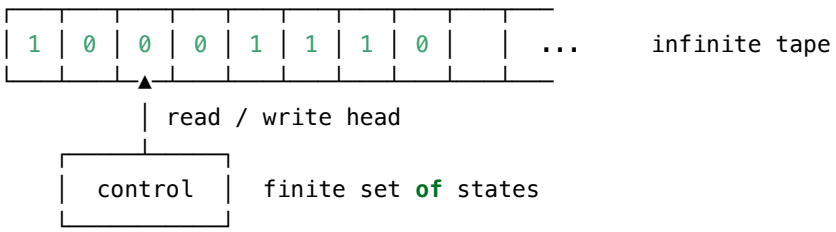
---

In class lecture notebook: https://gitlab.cs.umd.edu/cmsc330-mamat/fall2026/templates/lecture/lectures

---

### 1.3 Models of Computation

**Turing machine** (Alan Turing, 1936)




- A model of computation built on **state**: a tape that is read and overwritten

**Lambda calculus** (Alonzo Church, 1930s)

- A model of computation built on **functions**: no state, no tape — only defining and calling functions
- Both models compute exactly the same functions (the **Church–Turing thesis**)

---

## 1.4 Lambda Calculus ($\lambda$-calculus)

- Proposed in the 1930s by **Alonzo Church** (born in Washington, DC!)
- A formal system
  - Designed to investigate functions and recursion
  - For exploration of the foundations of mathematics
- Now used as
  - A tool for investigating computability
  - The basis of functional programming languages: Lisp, Scheme, ML, OCaml, Haskell, …

## 1.5 Why Study Lambda Calculus?

- It is a **core** language
  - Very small, but still **Turing complete**
- With it we can explore general ideas
  - Language features, semantics, proof systems, algorithms, …

---

## 1.6 Lambda Calculus Syntax

A lambda calculus **expression** is defined as

```
e ::= x          variable
    | λx.e       abstraction  (function definition)
    | e e        application  (function call)
```

- `λx.e` is like `(fun x -> e)` in OCaml
- That's it: no numbers, no booleans, no `if`, no `let`, no recursion

As an OCaml type — the same three cases:

[ ]:
```
type var = string;;

type exp =
  | Var of var             (* x    *)
  | Lam of var * exp       (* λx.e *)
  | App of exp * exp;;     (* e e  *)
```

---

| Lambda term | OCaml value |
|---|---|
| `y` | `Var "y"` |
| `λx.x` | `Lam ("x", Var "x")` |
| `λx.λy.x y` | `Lam ("x", Lam ("y", App (Var "x", Var "y")))` |
| `(λx.λy.x y) λx.x x` | `App (Lam ("x", Lam ("y", App (Var "x", Var "y"))), Lam ("x", App (Var "x", Var "x")))` |

---

## 1.7 Two Conventions

- The scope of $\lambda$ extends **as far right as possible**
  - Subject to scope delimited by parentheses
  - $\lambda$`x.`$\lambda$`y.x y` is the same as $\lambda$`x.(`$\lambda$`y.(x y))`
- Function application is **left-associative**
  - `x y z` is `(x y) z`
  - Same rule as OCaml

A printer that follows both conventions puts in only the parentheses that are needed:

[ ]:
```
let rec show (e : exp) : string =
  match e with
  | Var x -> x
  | Lam (x, body) -> "λ" ^ x ^ "." ^ show body     (* body extends far right *)
  | App (e1, e2) -> show_fun e1 ^ " " ^ show_arg e2

(* function position: a λ must be parenthesized, an application need not be (left-assoc)
 ↪*)
and show_fun e =
  match e with
  | Lam _ -> "(" ^ show e ^ ")"
  | _ -> show e

(* argument position: anything but a variable is parenthesized *)
and show_arg e =
  match e with
  | Var x -> x
  | _ -> "(" ^ show e ^ ");;
```

[ ]:
```
show (Lam ("x", Lam ("y", App (Var "x", Var "y"))));;
show (App (App (Var "x", Var "y"), Var "z"));;     (* (x y) z *)
show (App (Var "x", App (Var "y", Var "z")));;     (* x (y z) *)
```

---

## 1.8 Quiz 1: $\lambda$`x.x a b` is equivalent to which of the following?

- A. `(`$\lambda$`x.x) (a b)`
- B. `(((`$\lambda$`x.x) a) b)`
- C. $\lambda$`x.(x (a b))`
- D. `(`$\lambda$`x.((x a) b))`

Answer:

Answer: **D**. The $\lambda$`x` extends as far right as possible, so the body is `x a b`; and application is left-associative, so `x a b` is `(x a) b`.

```
[ ]: let quiz1 = Lam ("x", App (App (Var "x", Var "a"), Var "b"));;      (* λx.((x a) b) *)
     show quiz1;;
```

---

### 1.9 Lambda Calculus Semantics

- Evaluation of `(λx.e1) e2`: evaluate `e1` with `x` replaced by `e2`
- This is **beta-reduction** (substitution):

$$(\lambda x.e_1)\ e_2 \ \rightarrow\ e_1[x := e_2]$$

- $e_1[x := e_2]$ means "$e_1$ with every free `x` replaced by $e_2$"

---

### 1.10 Beta Reduction Example

`(λx.λz.x z) y  →  (λz.x z)[x := y]  =  λz.y z`

Equivalent OCaml code:

```
(fun x -> (fun z -> (x z))) y   →   fun z -> (y z)
```

---

### 1.11 Eager Evaluation (Call-by-Value, CBV)

Evaluate the argument $e_2$ **before** performing the beta-reduction — this is what OCaml does:

$$\frac{}{\lambda x.e_1 \Downarrow \lambda x.e_1} \qquad \frac{e_1 \Downarrow \lambda x.e_3 \qquad e_2 \Downarrow e_4 \qquad e_3[x := e_4] \Downarrow e_5}{e_1\ e_2 \Downarrow e_5}$$

### 1.12 Lazy Evaluation (Call-by-Name, CBN)

Perform the beta-reduction **without** evaluating $e_2$; use it as is:

$$\frac{}{\lambda x.e_1 \Downarrow \lambda x.e_1} \qquad \frac{e_1 \Downarrow \lambda x.e_3 \qquad e_3[x := e_2] \Downarrow e_4}{e_1\ e_2 \Downarrow e_4}$$

- In both, a function is already a value: we do not evaluate inside its body

```
(fun x -> 1) (1+2+3+....+1000000) = 1

eager:
(fun x -> 1) (infinite loop) stuck
lazy: (fun x -> 1) (infinite loop) = 1
```

---

### 1.13 Beta Reductions (CBV)

| | |
|---|---|
| (λx.x) z → z | |
| (λx.y) z → y | |
| (λx.x y) z → z y | a function that applies its argument to y |
| (λx.x y) (λz.z) → (λz.z) y → y | |
| (λx.λy.x y) z → λy.z y | a curried function of two arguments; applies its first argument to its second |
| (λx.λy.x y) (λz.z z) x → (λy.(λz.z z) y) x → (λz.z z) x → x x | |

To run these, we first need substitution. For now, a **naive** version — we will see in a few slides why it is not good enough:

[ ]:
```
(* naive: e[y := m] -- replace every y in e with m *)
let rec subst_naive (e : exp) (y : var) (m : exp) : exp =
  match e with
  | Var x -> if x = y then m else e
  | App (e1, e2) -> App (subst_naive e1 y m, subst_naive e2 y m)
  | Lam (x, body) ->
    if x = y then e     (* y is re-bound here: stop *)
    else Lam (x, subst_naive body y m);;
```

A **small-step** evaluator does one → step at a time. `step` returns `None` when no rule applies. A free variable such as `z` is treated as a value: it is "stuck", like a constant.

Call-by-value, left to right:

[ ]:
```
let is_value (e : exp) : bool =
  match e with
  | Lam _ | Var _ -> true
  | App _ -> false;;

let rec step_cbv (subst : exp -> var -> exp -> exp) (e : exp) : exp option =
  match e with
  | App (Lam (x, body), v) when is_value v ->
    Some (subst body x v)                                  (* β, argument is a value *)
  | App (e1, e2) when not (is_value e1) ->
    Option.map (fun e1' -> App (e1', e2)) (step_cbv subst e1)    (* function first *)
  | App (e1, e2) ->
    Option.map (fun e2' -> App (e1, e2')) (step_cbv subst e2)    (* then the argument *)
  | _ -> None;;
```

`trace` prints every step until no step applies (or until `max` steps — some terms never stop!):

[ ]:
```
let trace ?(max = 30) (step : exp -> exp option) (e : exp) : exp =
  print_endline ("  " ^ show e);
  let rec go n e =
    match step e with
    | None -> e
```

```
    | Some e' when n < max -> print_endline ("→ " ^ show e'); go (n + 1) e'
    | Some _ -> print_endline "→ ... (gave up)"; e
  in
  go 0 e;;
```

[ ]:
```
trace (step_cbv subst_naive) (App (Lam ("x", App (Var "x", Var "y")), Var "z"));;     (*␣
 ↪(λx.x y) z *)
```

[ ]:
```
(* (λx.λy.x y) (λz.z z) x *)
trace (step_cbv subst_naive)
  (App (App (Lam ("x", Lam ("y", App (Var "x", Var "y"))),
             Lam ("z", App (Var "z", Var "z"))),
        Var "x"));;
```

### 1.14 Quiz 2: `(λx.y) z` can be beta-reduced to

- A. `y`
- B. `y z`
- C. `z`
- D. cannot be reduced

Answer: **A**. The body `y` contains no `x`, so substituting for `x` changes nothing: `y[x := z] = y`.

[ ]:
```
trace (step_cbv subst_naive) (App (Lam ("x", Var "y"), Var "z"));;
```

### 1.15 Quiz 3: Which of the following reduces to `λz.z`?

- a) `(λy.λz.x) z`
- b) `(λz.λx.z) y`
- c) `(λy.y) (λx.λz.z) w`
- d) `(λy.λx.z) z (λz.z)`

Answer: **c**. `(λy.y) (λx.λz.z) w → (λx.λz.z) w → λz.z`. - a) gives `λz.x` - b) gives `λx.y` - d) gives `z`

[ ]:
```
let quiz3 = [
  App (Lam ("y", Lam ("z", Var "x")), Var "z");                                  (* a *)
  App (Lam ("z", Lam ("x", Var "z")), Var "y");                                  (* b *)
  App (App (Lam ("y", Var "y"), Lam ("x", Lam ("z", Var "z"))), Var "w");        (* c *)
  App (App (Lam ("y", Lam ("x", Var "z")), Var "z"), Lam ("z", Var "z")) ];;     (* d *)

List.iter (fun e -> ignore (trace (step_cbv subst_naive) e); print_newline ()) quiz3;;
```

---

### 1.16 CBN Reduction

Call-by-name: do the $\beta$ step as soon as the function is a $\lambda$, **without** reducing the argument first.

[ ]:
```
let rec step_cbn (subst : exp -> var -> exp -> exp) (e : exp) : exp option =
  match e with
```

```
  | App (Lam (x, body), e2) -> Some (subst body x e2)     (* β, argument as is *)
  | App (e1, e2) -> Option.map (fun e1' -> App (e1', e2)) (step_cbn subst e1)
  | _ -> None;;
```

The same term, two different orders:

- CBV: `(λz.z) ((λy.y) x) → (λz.z) x → x`
- CBN: `(λz.z) ((λy.y) x) → (λy.y) x → x`

[ ]:
```
let e = App (Lam ("z", Var "z"), App (Lam ("y", Var "y"), Var "x"));;
trace (step_cbv subst_naive) e;;
```

[ ]:
```
trace (step_cbn subst_naive) e;;
```

### 1.17 Beta Reductions (CBN)

Under CBN the argument is copied into the body unevaluated:

- `(λx.x (λy.y)) (u r) → u r (λy.y)`
- `(λx.(λw.x w)) (y z) → λw.y z w`

[ ]:
```
trace (step_cbn subst_naive)
  (App (Lam ("x", App (Var "x", Lam ("y", Var "y"))), App (Var "u", Var "r")));;
```

[ ]:
```
trace (step_cbn subst_naive)
  (App (Lam ("x", Lam ("w", App (Var "x", Var "w"))), App (Var "y", Var "z")));;
```

---

### 1.18 Static Scoping & Alpha Conversion

- Lambda calculus uses **static scoping**
- Consider `(λx.x (λx.x)) z → ?`
  - The rightmost `x` refers to the **second** binding
  - This is a function that takes its argument and applies it to the identity function
- This function is "the same" as `λx.x (λy.y)`
  - Renaming bound variables consistently preserves meaning
  - This is called **alpha-renaming** or **alpha conversion**
  - Ex. `λx.x = λy.y = λz.z` and `λy.λx.y = λz.λx.z`

[14]:
```
trace (step_cbv subst_naive)
  (App (Lam ("x", App (Var "x", Lam ("x", Var "x"))), Var "z"));;     (* (λx.x (λx.x)) z␣
  ↪*)
```

```
  (λx.x (λx.x)) z
→ z (λx.x)
```

[14]:
```
- : exp = App (Var "z", Lam ("x", Var "x"))
```

### 1.19 Quiz 4: Which of the following expressions is alpha equivalent to `(λx.λy.x y) y`?

- a) `λy.y y`
- b) `λz.y z`
- c) `(λx.λz.x z) y`
- d) `(λx.λy.x y) z`

int foo(int a){ int y = 0; { int x = 10; y = x + y; } return y + a; } foo 20

Answer:

Answer: **c**. It only renames the bound `y` to `z`. - a) and b) are reductions, not renamings - d) changes the **free** variable `y` — a different term

---

### 1.20 Getting Serious about Substitution

We have been thinking informally about substitution, but the details matter. Let's carefully formalize $e_1[x := e_2]$, to see where it can get tricky.

| Case | $(\lambda x.e_1)\ e_2 \to e_1[x := e_2]$ | |
|---|---|---|
| 1 | `(λx.x) e2 → x[x:=e2] = e2` | replace `x` by `e2` |
| 2 | `(λx.y) e2 → y[x:=e2] = y` | `y` is different than `x`, so no effect |
| 3 | `(λx.e0 e1) e2 → (e0[x:=e2]) (e1[x:=e2])` | substitute both parts of an application |
| 4 | `(λx.(λx.e')) e2 → (λx.e')[x:=e2] = λx.e'` | the inner `x` **shadows** the outer: stop |
| 5 | `(λx.(λy.e')) e2 → (λy.e')[x:=e2] = ?` | … |

Example of case 4: `(λx.(λx.x)) a → λx.x`

---

### 1.21 Case 5: Substituting Under a Different Binder

`(λx.(λy.e')) e2 → (λy.e')[x:=e2]`

$(\lambda y.e')[x := e_2]$ where $y \neq x$:

- If $y \notin FV(e_2)$: just go inside, $\lambda y.(e'[x := e_2])$
  - `(λx.λy.x y) z → λy.z y`
- If $y \in FV(e_2)$: going inside would **capture** the free `y` in $e_2$!
  - `(λx.λy.x y) y → λy.y y` **wrong!** The free `y` became bound
  - Instead, first alpha-convert $y$ to a **fresh** variable $w$ that does not appear in $e'$ or $e_2$
  - `(λx.λy.x y) y = (λx.λz.x z) y → λz.y z`

Formally: $(\lambda y.e')[x := e_2] = \lambda w.((e'[y := w])[x := e_2])$, $w$ fresh

Our naive substitution gets this wrong:

[15]:
```
let capture = App (Lam ("x", Lam ("y", App (Var "x", Var "y"))), Var "y");;     (* (λx.
  ↪λy.x y) y *)
trace (step_cbv subst_naive) capture;;     (* wrong: free y was captured *)
```

[15]:
```
val capture : exp =
  App (Lam ("x", Lam ("y", App (Var "x", Var "y"))), Var "y")


  (λx.λy.x y) y
→ λy.y y
```

[15]:
```
- : exp = Lam ("y", App (Var "y", Var "y"))
```

---

## 1.22 Free Variables

$$FV(x) = \{x\}$$
$$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$$
$$FV(\lambda x.e) = FV(e) - \{x\}$$

Examples: `- FV(x) = {x}`

- `FV(x y) = {x, y}`
- `FV(λx.x) = FV(x) - {x} = {}`
- `FV(λx.x y) = FV(x y) - {x} = {y}`
- `FV((λx.x y) x) = FV(λx.x y) ∪ FV(x) = {x, y}`

[16]:
```
(* computes the free (non-bound) variables in e *)
let rec fvs (e : exp) : var list =
  match e with
  | Var x -> [ x ]
  | Lam (x, e) -> List.filter (fun y -> x <> y) (fvs e)
  | App (e1, e2) -> fvs e1 @ fvs e2;;

fvs (Lam ("x", App (Var "x", Var "y")));;                    (* λx.x y *)
fvs (App (Lam ("x", App (Var "x", Var "y")), Var "x"));;     (* (λx.x y) x *)
```

[16]:
```
val fvs : exp -> var list = <fun>
```

[16]:
```
- : var list = ["y"]
```

[16]:
```
- : var list = ["y"; "x"]
```

---

### 1.23 OCaml Implementation: Substitution

`subst e y m` computes $e[y := m]$ — substitute `m` for `y` in `e`:

[18]:
```
(* generates a fresh variable name *)
let newvar =
  let x = ref 0 in
  fun () ->
    let c = !x in
    incr x;
    "v" ^ string_of_int c;;

(* substitution: subst e y m means
   "substitute occurrences of variable y with m in the expression e" *)
let rec subst (e : exp) (y : var) (m : exp) : exp =
  match e with
  | Var x ->
    if y = x then m      (* replace x with m *)
    else e               (* variables don't match: leave x alone *)
  | App (e1, e2) -> App (subst e1 y m, subst e2 y m)
  | Lam (x, e) ->
    if y = x then        (* shadowing blocks substitution *)
      Lam (x, e)
    else if not (List.mem x (fvs m)) then
      Lam (x, subst e y m)       (* safe: no capture possible *)
    else
      (* might capture; need to alpha convert *)
      let z = newvar () in       (* assumed to be "fresh" *)
      let e' = subst e x (Var z) in
      Lam (z, subst e' y m);;
```

[18]:
```
val newvar : unit -> string = <fun>
```

[18]:
```
val subst : exp -> var -> exp -> exp = <fun>
```

[19]:
```
trace (step_cbv subst) capture;;     (* (λx.λy.x y) y → λv0.y v0 *)
```

```
  (λx.λy.x y) y
→ λv0.y v0
```

[19]:
```
- : exp = Lam ("v0", App (Var "y", Var "v0"))
```

## 1.24 Reducing to Normal Form

- A **normal form** is a term with no beta-redex anywhere — it cannot be reduced further
- CBV and CBN stop at a $\lambda$: they never reduce inside a function body
- To compare terms such as Church numerals (coming up), we want to reduce **everywhere**, including under $\lambda$

`reduce` does one step: a $\beta$ step at the top if possible, otherwise it tries the left side of an application, then the right side, and it also reduces inside function bodies:

[ ]:
```
(* beta reduction: one step, leftmost-outermost first *)
let rec reduce (e : exp) : exp =
  match e with
  | App (Lam (x, e), e2) -> subst e x e2      (* direct beta rule *)
  | App (e1, e2) ->
    let e1' = reduce e1 in                    (* try to reduce a term in the lhs *)
    if e1' <> e1 then App (e1', e2)
    else App (e1, reduce e2)                  (* didn't work; try rhs *)
  | Lam (x, e) -> Lam (x, reduce e)           (* reduce under the lambda (!) *)
  | _ -> e;;                                  (* no opportunity to reduce *)

(* reduce until nothing changes: the normal form *)
let rec reduce_multi (e : exp) : exp =
  let e' = reduce e in
  if e' = e then e else reduce_multi e';;

(* one step as an option, for trace *)
let step_full (e : exp) : exp option =
  let e' = reduce e in
  if e' = e then None else Some e';;
```

- The $\beta$ rule fires **before** the argument is reduced, so the argument is substituted as is: this is the **lazy** (call-by-name) order, extended to reduce inside bodies. It is called **normal order** reduction
- Normal order has a nice property: if a term has a normal form, normal order finds it
- `e1' <> e1` is **structural** inequality. With physical inequality `!=` (as in some versions of this code), `Lam`/`App` always rebuild a new node, so `x y ((λz.z) w)` would stop before reducing `(λz.z) w`

[ ]:
```
trace step_full (App (App (Var "x", Var "y"), App (Lam ("z", Var "z"), Var "w")));;
↪(* x y ((λz.z) w) *)
```

## 1.25 The Power of Lambdas

To give a sense of how one can encode various constructs into the lambda calculus, we'll look at some concrete examples:

- Let bindings
- Booleans
- Pairs
- Natural numbers & arithmetic
- Looping

Encodings use multi-argument applications a lot, so one helper: `apps f [a; b; c]` builds `f a b c`.

[ ]:
```
let apps (f : exp) (args : exp list) : exp =
  List.fold_left (fun acc a -> App (acc, a)) f args;;

show (apps (Var "f") [ Var "a"; Var "b"; Var "c" ]);;
```

## 1.26 Let Bindings

Local variable declarations are like defining a function and applying it immediately (once):

`let x = e1 in e2  =  (λx.e2) e1`

Example: `let x = (λy.y) in x x = (λx.x x) (λy.y)`

`(λx.x x) (λy.y) → (λy.y) (λy.y) → λy.y`

[ ]:
```
let let_ (x : var) (e1 : exp) (e2 : exp) : exp = App (Lam (x, e2), e1);;

(* let x = (λy.y) in x x *)
trace step_full (let_ "x" (Lam ("y", Var "y")) (App (Var "x", Var "x")));;
```

## 1.27 Booleans

Church's encoding of mathematical logic:

- `true  = λx.λy.x` — pick the first
- `false = λx.λy.y` — pick the second
- `if a then b else c` is defined to be the expression `a b c`

Examples:

- `if true then b else c = (λx.λy.x) b c → (λy.b) c → b`
- `if false then b else c = (λx.λy.y) b c → (λy.y) c → c`

[ ]:
```
let mytrue = Lam ("x", Lam ("y", Var "x"));;      (* λx.λy.x *)
let myfalse = Lam ("x", Lam ("y", Var "y"));;     (* λx.λy.y *)
let myif = Lam ("a", Lam ("b", Lam ("c", apps (Var "a") [ Var "b"; Var "c" ])));;     (*␣
 ↪λa.λb.λc.a b c *)

trace step_full (apps mytrue [ Var "b"; Var "c" ]);;
```

```
[ ]: trace step_full (apps myif [ myfalse; Var "b"; Var "c" ]);;
```

## 1.28 Booleans (cont.)

Other Boolean operations:

- not = λx.x false true
  - not x = x false true = if x then false else true
  - not true → (λx.x false true) true → true false true → false
- and = λx.λy.x y false
  - and x y = if x then y else false
- or = λx.λy.x true y
  - or x y = if x then true else y

Given these operations, we can build up a logical inference system.

(lambda/src/encoding.ml writes these differently, e.g. and = λa.λb.λx.λy.b (a x y) y, but they compute the same results.)

```
[ ]: let mynot = Lam ("x", apps (Var "x") [ myfalse; mytrue ]);;                    (* λx.x false␣
      ↪true *)
     let myand = Lam ("x", Lam ("y", apps (Var "x") [ Var "y"; myfalse ]));;      (* λx.λy.x y␣
      ↪false *)
     let myor = Lam ("x", Lam ("y", apps (Var "x") [ mytrue; Var "y" ]));;        (* λx.λy.x␣
      ↪true y *)

     (* decode a Church boolean back into an OCaml bool *)
     let to_bool (e : exp) : bool =
       match reduce_multi e with
       | Lam (x, Lam (_, Var z)) when z = x -> true
       | Lam (_, Lam (y, Var z)) when z = y -> false
       | e' -> failwith ("not a boolean: " ^ show e');;
```

```
[ ]: trace step_full (App (mynot, mytrue));;
```

```
[ ]: List.map (fun (a, b) -> to_bool (apps myand [ a; b ]))
       [ (mytrue, mytrue); (mytrue, myfalse); (myfalse, mytrue); (myfalse, myfalse) ];;
```

```
[ ]: List.map (fun (a, b) -> to_bool (apps myor [ a; b ]))
       [ (mytrue, mytrue); (mytrue, myfalse); (myfalse, mytrue); (myfalse, myfalse) ];;
```

## 1.29 Pairs

Encoding of a pair a, b:

- (a, b) = λx.if x then a else b
- fst = λf.f true
- snd = λf.f false

Examples:

- `fst (a, b) = (λf.f true) (λx.if x then a else b) → (λx.if x then a else b) true → if true then a else b → a`
- `snd (a, b) = (λf.f false) (λx.if x then a else b) → (λx.if x then a else b) false → if false then a else b → b`

[ ]:
```
(* (a, b) = λx.if x then a else b = λx.x a b *)
let mypair (a : exp) (b : exp) : exp = Lam ("x", apps (Var "x") [ a; b ]);;
let myfst = Lam ("f", App (Var "f", mytrue));;      (* λf.f true *)
let mysnd = Lam ("f", App (Var "f", myfalse));;     (* λf.f false *)

trace step_full (App (myfst, mypair (Var "a") (Var "b")));;
```

[ ]:
```
reduce_multi (App (mysnd, mypair (Var "a") (Var "b")));;
```

---

### 1.30 Natural Numbers (Church Numerals)

Encoding of non-negative integers:

- `0 = λf.λx.x`
- `1 = λf.λx.f x`
- `2 = λf.λx.f (f x)`
- `3 = λf.λx.f (f (f x))`
- i.e., `n = λf.λx.<apply f n times to x>`
- Formally: `n+1 = λf.λx.f (n f x)`

The number $n$ **is** "do something $n$ times." In OCaml:

[ ]:
```
let two f x = f (f x);;
two (fun n -> n + 1) 0;;            (* add 1, twice *)
two (fun s -> s ^ "!") "hi";;       (* append "!", twice *)
```

In our lambda calculus implementation:

[ ]:
```
let myzero = Lam ("f", Lam ("x", Var "x"));;
let myone = Lam ("f", Lam ("x", App (Var "f", Var "x")));;
let mytwo = Lam ("f", Lam ("x", App (Var "f", App (Var "f", Var "x"))));;
let mythree = Lam ("f", Lam ("x", App (Var "f", App (Var "f", App (Var "f", Var "x")))));;

(* church n = λf.λx.f (f (... (f x))), n times *)
let church (n : int) : exp =
  let rec body n = if n = 0 then Var "x" else App (Var "f", body (n - 1)) in
  Lam ("f", Lam ("x", body n));;

(* decode: reduce to normal form, then count the f's *)
let to_int (e : exp) : int =
  match reduce_multi e with
  | Lam (f, Lam (x, body)) ->
    let rec count e =
```

```
      match e with
      | Var y when y = x -> 0
      | App (Var g, e') when g = f -> 1 + count e'
      | _ -> failwith ("not a numeral: " ^ show e)
    in
    count body
  | e' -> failwith ("not a numeral: " ^ show e');;

show (church 4);;
to_int mythree;;
```

## 1.31 Operations on Church Numerals

Successor:

- succ = λn.λf.λx.f (n f x)

Example:

```
succ 0 = (λn.λf.λx.f (n f x)) (λf.λx.x)
       → λf.λx.f ((λf.λx.x) f x)
       → λf.λx.f ((λx.x) x)
       → λf.λx.f x
       = 1
```

[ ]:
```
let mysucc = Lam ("n", Lam ("f", Lam ("x", App (Var "f", apps (Var "n") [ Var "f"; Var
  ↪"x" ])))));;

trace step_full (App (mysucc, myzero));;
```

## 1.32 Operations on Church Numerals (cont.)

IsZero?

- iszero = λn.n (λx.false) true
  - Equivalent to λn.((n (λx.false)) true)
  - Apply "always false" $n$ times to true: zero times leaves true

Example:

```
iszero 0 = (λn.n (λx.false) true) (λf.λx.x)
         → (λf.λx.x) (λx.false) true
         → (λx.x) true
         → true
```

[ ]:
```
let iszero = Lam ("n", apps (Var "n") [ Lam ("x", myfalse); mytrue ]);;

trace step_full (App (iszero, myzero));;
```

[ ]:
```
to_bool (App (iszero, mytwo));;
```

### 1.33 Arithmetic Using Church Numerals

If `M` and `N` are numbers (as $\lambda$ expressions), we can also encode arithmetic:

- Addition — apply `f N` times, then `M` more times
  - `M + N = λf.λx.M f (N f x)`
  - Equivalently: `+ = λM.λN.λf.λx.M f (N f x)`, in prefix notation `(+ M N)`
- Multiplication — repeat "apply `f N` times", `M` times
  - `M * N = λf.M (N f)`
  - Equivalently: `* = λM.λN.λf.M (N f)`, in prefix notation `(* M N)`

[ ]:
```
(* λm.λn.λf.λx.m f (n f x) *)
let myplus =
  Lam ("m", Lam ("n", Lam ("f", Lam ("x",
    apps (Var "m") [ Var "f"; apps (Var "n") [ Var "f"; Var "x" ] ]))));;

(* λm.λn.λf.m (n f) *)
let mymult = Lam ("m", Lam ("n", Lam ("f", App (Var "m", App (Var "n", Var "f")))));;
```

### 1.34 Arithmetic (cont.)

Prove `1 + 1 = 2`:

[ ]:
```
trace step_full (apps myplus [ myone; myone ]);;
```

With these definitions, we can build a theory of arithmetic:

[ ]:
```
to_int (apps myplus [ mythree; mytwo ]);;     (* 3 + 2 *)
```

[ ]:
```
(* (3 + 2) * succ 3 *)
let five = apps myplus [ mythree; mytwo ];;
let four = App (mysucc, mythree);;
to_int (apps mymult [ five; four ]);;
```

### 1.35 Subtraction

- Easy once you have **predecessor**, but predecessor is very difficult!
- Story time: one of Church's students, **Kleene** (of Kleene-star fame), was struggling to think of how to encode predecessor, until it came to him during a trip to the dentist's office
  - Take from this what you will

```
pred  = λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)
minus = λm.λn.n pred m            (* apply pred to m, n times *)
```

[ ]:
```
let mypred =
  Lam ("n", Lam ("f", Lam ("x",
    apps (Var "n") [ Lam ("g", Lam ("h", App (Var "h", App (Var "g", Var "f"))));
                     Lam ("u", Var "x");
                     Lam ("u", Var "u") ])));;

let myminus = Lam ("m", Lam ("n", apps (Var "n") [ mypred; Var "m" ]));;
```

```
to_int (App (mypred, mythree));;
```

[ ]:
```
to_int (apps myminus [ five; mytwo ]);;     (* (3 + 2) - 2 *)
```

With `if`, `and`, and numbers we can already write conditionals over numbers:

[ ]:
```
(* if (and true true) then 1 else 2 *)
to_int (apps myif [ apps myand [ mytrue; mytrue ]; myone; mytwo ]);;
```

[ ]:
```
(* if (and true false) then 1 else 2 *)
to_int (apps myif [ apps myand [ mytrue; myfalse ]; myone; mytwo ]);;
```

---

### 1.36 Looping + Recursion

- So far we have avoided self-reference, so how does recursion work?
- We can construct a lambda term that "replicates" itself:
  - Define `D = λx.x x`, then
  - `D D = (λx.x x) (λx.x x) → (λx.x x) (λx.x x) = D D`
  - `D D` is an infinite loop
- We want to generalize this, so that we can make use of looping

[ ]:
```
let d = Lam ("x", App (Var "x", Var "x"));;
trace ~max:3 (step_cbn subst) (App (d, d));;     (* D D → D D → D D → ... *)
```

### 1.37 The Fixpoint Combinator

`Y = λf.(λx.f (x x)) (λx.f (x x))`

Then

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

- `Y F` is a **fixed point** (aka fixpoint) of `F`
- Thus `Y F = F (Y F) = F (F (Y F)) = ...`
  - We can use `Y` to achieve recursion for `F`

[ ]:
```
let yfix =
  Lam ("f", App (Lam ("x", App (Var "f", App (Var "x", Var "x"))),
                 Lam ("x", App (Var "f", App (Var "x", Var "x")))));;

trace ~max:5 step_full (App (yfix, Var "F"));;     (* F (F (F ...)) forever *)
```

### 1.38 Example: Factorial

`fact = λf.λn.if n = 0 then 1 else n * (f (n-1))`

- The second argument to `fact` is the integer
- The first argument is the function to call in the body
  - We'll use `Y` to make this recursively call `fact`

```
(Y fact) 1 = (fact (Y fact)) 1
  → if 1 = 0 then 1 else 1 * ((Y fact) 0)
  → 1 * ((Y fact) 0)
  = 1 * (fact (Y fact) 0)
  → 1 * (if 0 = 0 then 1 else 0 * ((Y fact) (-1)))
  → 1 * 1
  → 1
```

[ ]:
```
let if2 (a, b, c) = apps myif [ a; b; c ];;

(* λf.λn.if iszero n then 1 else n * (f (pred n)) *)
let fact1 =
  Lam ("f", Lam ("n",
    if2 ( App (iszero, Var "n"),                                    (* condition *)
          myone,                                                    (* if branch *)
          apps mymult [ Var "n"; App (Var "f", App (mypred, Var "n")) ] )));;   (* else *)

to_int (apps yfix [ fact1; mythree ]);;     (* fact(3) *)
```

[ ]:
```
to_int (apps yfix [ fact1; church 4 ]);;     (* fact(4) *)
```

### 1.39 Factorial 4 = ?

Writing `G` for `fact`:

```
(Y G) 4
G (Y G) 4
(λr.λn.if n = 0 then 1 else n × (r (n-1))) (Y G) 4
(λn.if n = 0 then 1 else n × ((Y G) (n-1))) 4
if 4 = 0 then 1 else 4 × ((Y G) (4-1))
4 × (G (Y G) (4-1))
4 × (3 × (G (Y G) (3-1)))
4 × (3 × (2 × (G (Y G) (2-1))))
4 × (3 × (2 × (1 × (G (Y G) (1-1)))))
4 × (3 × (2 × (1 × (if 0 = 0 then 1 else 0 × ((Y G) (0-1)))))))
4 × (3 × (2 × (1 × 1)))
24
```

`Y` only works because our reducer is **lazy**: it substitutes `Y G` without reducing it first. Under call-by-value, `Y G` would be reduced before every call — forever.

[ ]:
```
trace ~max:4 (step_cbv subst) (App (yfix, Lam ("r", Var "r")));;     (* CBV: never␣
  ↪reaches the call *)
```

---

## 1.40 Discussion

- Lambda calculus is **Turing-complete**
  - The most powerful language possible
  - Can represent pretty much anything in a "real" language, using clever encodings
- But programs would be
  - Pretty slow (`10000 + 1` → thousands of function calls)
  - Pretty large (`10000 + 1` → hundreds of lines of code)
  - Pretty hard to understand (recognize `10000` vs. `9999`)
- In practice
  - We use richer, more **expressive** languages
  - That include built-in primitives

[ ]:
```
let rec size (e : exp) : int =
  match e with
  | Var _ -> 1
  | Lam (_, e) -> 1 + size e
  | App (e1, e2) -> 1 + size e1 + size e2;;

size (church 10000);;     (* AST nodes just to write down 10000 *)
```

## 1.41 The Need for Types

- Consider the **untyped** lambda calculus
  - `false = λx.λy.y`
  - `0 = λf.λx.x`
- Since everything is encoded as a function, we can easily misuse terms
  - `false 0 → λy.y`
  - `if 0 then ...`
  - … because everything evaluates to some function
- The same thing happens in assembly language
  - Everything is a machine word (a bunch of bits)
  - All operations take machine words to machine words

[ ]:
```
(show myfalse, show myzero);;     (* the same term, up to renaming *)
reduce_multi (App (myfalse, myzero));;     (* false 0: no error, just some function *)
```

[ ]:
```
show (reduce_multi (apps myplus [ mytrue; mytwo ]));;     (* true + 2: some function, no␣
 ↪error *)
```

## 1.42 Simply-Typed Lambda Calculus (STLC)

```
e ::= n | x | λx:t.e | e e
t ::= int | t → t
```

- Added integers `n` as primitives
  - Need at least two distinct types (integer & function) to have type errors
- Functions now include the type `t` of their argument
- `int` is the type of integers; `t1 → t2` is the type of a function that takes arguments of type `t1` and returns a result of type `t2`

This is the `Fun (x, t, e)` / `TArrow` fragment of the type checker from Lecture 16.

### 1.43 Types Are Limiting

- STLC will reject some terms as ill-typed, even if they will not produce a run-time error
  - Cannot type check `Y` in STLC
  - Or in OCaml, for that matter, at least not as written earlier — `x x` needs `x` to have type `t` and `t → t'` at the same time:

[ ]:
```
let d = fun x -> x x;;     (* Error: x would need type 'a and 'a -> 'b *)
```

- Surprising theorem: all (well typed) simply-typed lambda calculus terms are **strongly normalizing**
  - A **normal form** is one that cannot be reduced further; a **value** is a kind of normal form
  - Strong normalization means STLC terms **always terminate**
  - The proof is *not* by straightforward induction: applications "increase" term size
- So `D D` and `Y` cannot be typed: a language with only STLC can't loop. Real languages add recursion back as a primitive — OCaml's `let rec`

---

### 1.44 Summary

- Lambda calculus is a core model of computation
  - `e ::= x | λx.e | e e`, evaluated by **beta-reduction** $(\lambda x.e_1)\ e_2 \to e_1[x := e_2]$
  - Substitution must avoid **capturing** free variables: **alpha-convert** to a fresh name
  - Evaluation order: **eager** (CBV) vs. **lazy** (CBN); normal order reduces under $\lambda$ too
- We can encode familiar language constructs using only functions
  - `let`, booleans, pairs, natural numbers & arithmetic, recursion via `Y`
  - These encodings are enlightening – they make you a better (functional) programmer
- Useful for understanding how languages work
  - Ideas of types, evaluation order, termination, proof systems, etc. can be developed in lambda calculus, then scaled to full languages