Lambda Calculus : Encodings

CS3100 Fall 2019

Power of Lambdas

- Despite its simplicity, the lambda calculus is quite expressive: it is **Turing complete**!
- Means we can encode any computation we want
  - if we are sufficiently clever...
- Examples
  - Booleans & predicate logic.
  - Pairs
  - Lists
  - Natural numbers & arithmetic.

In [1]:

```
#use "init.ml"

let p = Lambda_parse.parse_string
let var x = Var x
let app l =
  match l with
  | [] -> failwith "ill typed app"
  | [x] -> x
  | x::y::xs -> List.fold_left (fun expr v -> App (expr, v)) (App(x,y))
let lam x e = Lam(x,e)

let eval ?(log=true) ?(depth=1000) s =
     s
  |> Eval.eval ~log ~depth Eval.reduce_normal
  |> Syntax.string_of_expr
```

```
Findlib has been successfully loaded. Additional directive
s:
  #require "package";;      to load a package
  #list;;                   to list the available packages
  #camlp4o;;                to load camlp4 (standard synta
x)
  #camlp4r;;                to load camlp4 (revised syntax)
  #predicates "p,q,...";;   to set these predicates
  Topfind.reset();;         to force that packages will be
reloaded
  #thread;;                 to enable threads
```

Out[1]:

```
val p : string -> Syntax.expr = <fun>
```

Out[1]:

```
val var : string -> Syntax.expr = <fun>
```

Out[1]:

```
val app : Syntax.expr list -> Syntax.expr = <fun>
```

Out[1]:

```
val lam : string -> Syntax.expr -> Syntax.expr = <fun>
```

Out[1]:

```
val eval : ?log:bool -> ?depth:int -> Syntax.expr -> string
= <fun>
```

Booleans

In [2]:

```
let tru = p "\\t.\\f.t"
let fls = p "\\t.\\f.f"
```

Out[2]:

```
val tru : Syntax.expr = Lam ("t", Lam ("f", Var "t"))
```

Out[2]:

```
val fls : Syntax.expr = Lam ("t", Lam ("f", Var "f"))
```

- Now we can define a `test` function such that
  - `test tru v w` $\rightarrow_\beta$ `v`
  - `test fls v w` $\rightarrow_\beta$ `w`

In [3]:

```
let test = p "\\l.\\m.\\n.l m n"
```

Out[3]:

```
val test : Syntax.expr =
  Lam ("l", Lam ("m", Lam ("n", App (App (Var "l", Var
"m"), Var "n"))))
```

Booleans

Now

```
test tru v w
```

evaluates to

In [4]:

```
eval @@ app [test; tru; var "v"; var "w"]
```

```
= (λm.λn.(λt.λf.t) m n) v w
= (λn.(λt.λf.t) v n) w
= (λt.λf.t) v w
= (λf.v) w
= v
```

Out[4]:

```
- : string = "v"
```

Booleans

Similarly,

```
test fls v w
```

evaluates to

In [5]:

```
eval @@ app [test; fls; var "v"; var "w"]
```

```
= (λm.λn.(λt.λf.f) m n) v w
= (λn.(λt.λf.f) v n) w
= (λt.λf.f) v w
= (λf.f) w
= w
```

Out[5]:

```
- : string = "w"
```

Booleans

`fls` itself is a function. `test fls v w` is equivalent to `fls v w`.

In [6]:

```
eval @@ app [fls; var "v"; var "w"]
```

```
= (λf.f) w
= w
```

Out[6]:

```
- : string = "w"
```

Logical operators

```
and = λb.λc.b c fls
or = λb.λc.b tru c
not = λb.b fls tru
```

In [7]:

```
let and_ = lam "b" (lam "c" (app [var "b"; var "c"; fls]))
let or_ = lam "b" (lam "c" (app [var "b"; tru; var "c"]))
let not_ = lam "b" (app [var "b"; fls; tru])
```

Out[7]:

```
val and_ : Syntax.expr =
  Lam ("b",
   Lam ("c", App (App (Var "b", Var "c"), Lam ("t", Lam
("f", Var "f")))))
```

Out[7]:

```
val or_ : Syntax.expr =
  Lam ("b",
   Lam ("c", App (App (Var "b", Lam ("t", Lam ("f", Var
"t"))), Var "c")))
```

Out[7]:

```
val not_ : Syntax.expr =
  Lam ("b",
   App (App (Var "b", Lam ("t", Lam ("f", Var "f"))),
    Lam ("t", Lam ("f", Var "t"))))
```

Logical Operators

In [8]:

```
eval @@ app [and_; tru; fls]
```

```
= (λc.(λt.λf.t) c (λt.λf.f)) (λt.λf.f)
= (λt.λf.t) (λt.λf.f) (λt.λf.f)
= (λf.λt.λf.f) (λt.λf.f)
= λt.λf.f
```

Out[8]:

```
- : string = "λt.λf.f"
```

The above is a **proof** for `true /\ false = false`

Logical operators

$$p \implies q \equiv \neg p \vee q$$

$$\textbf{Theorem1.} \quad a \wedge b \implies a$$

In [9]:

```
let implies = lam "p" (lam "q" (app [or_; app [not_; var "p"]; var "q"])
let thm1 = lam "a" (lam "b" (app [implies; app [and_; var "a"; var "b"];
```

Out[9]:

```
val implies : Syntax.expr =
  Lam ("p",
   Lam ("q",
    App
     (App
       (Lam ("b",
         Lam ("c",
          App (App (Var "b", Lam ("t", Lam ("f", Var
"t"))), Var "c"))),
       App
        (Lam ("b",
          App (App (Var "b", Lam ("t", Lam ("f", Var
"f"))),
           Lam ("t", Lam ("f", Var "t")))),
        Var "p")),
     Var "q")))
```

Out[9]:

```
val thm1 : Syntax.expr =
  Lam ("a",
   Lam ("b",
    App
     (App
       (Lam ("p",
         Lam ("q",
          App
           (App
             (Lam ("b",
               Lam ("c",
                App (App (Var "b", Lam ("t", Lam ("f", Var
"t"))), Var "c"))),
             App
              (Lam ("b",
                App (App (Var "b", Lam ("t", Lam ("f", Var
"f"))),
                 Lam ("t", Lam ("f", Var "t")))),
              Var "p")),
           Var "q"))),
       App
        (App
          (Lam ("b",
            Lam ("c",
             App (App (Var "b", Var "c"), Lam ("t", Lam
```

```
("f", Var "f"))))),
          Var "a"),
        Var "b")),
     Var "a")))
```

Logical operators

In [10]:

```
eval ~log:false (app [thm1; var "x"; var "y"])
```

Out[10]:

```
- : string = "x y (λt.λf.f) (λt.λf.f) (λt.λf.t) (λt.λf.t) x"
```

Quiz

What is the lambda calculus encoding for `xor x y`

x	y	xor x y
T	T	F
T	F	T
F	T	T
F	F	F

1. x x y
2. x (y tru fls) y
3. x (y fls tru) y
4. y x y

Quiz

What is the lambda calculus encoding for `xor x y`

x	y	xor x y
T	T	F
T	F	T
F	T	T
F	F	F

1. x x y
2. x (y tru fls) y
3. x (y fls tru) y ✅
4. y x y

Pairs

- Encoding of a pair `(a,b)`
  - (a,b) = λf.λs.λb.b f s
  - fst = λf.f tru
  - snd = λf.f fls

In [11]:

```
let pair = p ("λf.λs.λb.b f s")
let fst = lam "p" (app [var "p"; tru])
let snd = lam "p" (app [var "p"; fls])
```

Out[11]:

```
val pair : Syntax.expr =
  Lam ("f", Lam ("s", Lam ("b", App (App (Var "b", Var
"f"), Var "s"))))
```

Out[11]:

```
val fst : Syntax.expr =
  Lam ("p", App (Var "p", Lam ("t", Lam ("f", Var "t"))))
```

Out[11]:

```
val snd : Syntax.expr =
  Lam ("p", App (Var "p", Lam ("t", Lam ("f", Var "f"))))
```

Pairs

In [12]:

```
eval @@ app [fst; app [pair; var "v"; var "w"]]
```

```
= (λf.λs.λb.b f s) v w (λt.λf.t)
= (λs.λb.b v s) w (λt.λf.t)
= (λb.b v w) (λt.λf.t)
= (λt.λf.t) v w
= (λf.v) w
= v
```

Out[12]:

```
- : string = "v"
```

Natural numbers

- 0 = λs.λz.z
- 1 = λs.λz.s z
- 2 = λs.λz.s (s z)
- 3 = λs.λz.s (s (s z))

i.e., n = λs.λz.(apply `s` n times to `z`)

Also known as **Church numerals.**

Natural numbers

In [13]:

```
let zero = p ("λs.λz.z")
let one = p ("λs.λz.s z")
let two = p ("λs.λz.s (s z)")
let three = p ("λs.λz.s (s (s z))")
```

Out[13]:

```
val zero : Syntax.expr = Lam ("s", Lam ("z", Var "z"))
```

Out[13]:

```
val one : Syntax.expr = Lam ("s", Lam ("z", App (Var "s", V
ar "z")))
```

Out[13]:

```
val two : Syntax.expr =
  Lam ("s", Lam ("z", App (Var "s", App (Var "s", Var
"z"))))
```

Out[13]:

```
val three : Syntax.expr =
  Lam ("s", Lam ("z", App (Var "s", App (Var "s", App (Var
"s", Var "z")))))
```

Quiz

What will be the OCaml type of church encoded numeral?

1. `('a -> 'b) -> 'a -> 'b`
2. `('a -> 'a) -> 'a -> 'a`
3. `('a -> 'a) -> 'b -> int`
4. `(int -> int) -> int -> int`

Quiz

What will be the OCaml type of church encoded numeral?

1. `('a -> 'b) -> 'a -> 'b`
2. `('a -> 'a) -> 'a -> 'a` ✅
3. `('a -> 'a) -> 'b -> int`
4. `(int -> int) -> int -> int`

Operations on numbers: Successor

Successor function is:

```
scc = λn.λs.λz.s (n s z)
```

In [14]:

```
let scc = p ("λn.λs.λz.s (n s z)")
```

Out[14]:

```
val scc : Syntax.expr =
  Lam ("n",
   Lam ("s", Lam ("z", App (Var "s", App (App (Var "n", Var
"s"), Var "z")))))
```

In [15]:

```
eval @@ app [scc; zero]
```

```
= λs.λz.s ((λs.λz.z) s z)
= λs.λz.s ((λz.z) z)
= λs.λz.s z
```

Out[15]:

```
- : string = "λs.λz.s z"
```

Operations on numbers : is_zero

Check if the given number is zero:

```
is_zero = λn.n (λy.fls) tru
```

In [16]:

```
let is_zero = lam "n" (app [var "n"; lam "y" fls; tru])
```

Out[16]:

```
val is_zero : Syntax.expr =
  Lam ("n",
   App (App (Var "n", Lam ("y", Lam ("t", Lam ("f", Var
"f")))),
    Lam ("t", Lam ("f", Var "t"))))
```

Operations on numbers : is_zero

In [17]:

```
eval @@ app [is_zero; zero]
```

```
= (λs.λz.z) (λy.λt.λf.f) (λt.λf.t)
= (λz.z) (λt.λf.t)
= λt.λf.t
```

Out[17]:

```
- : string = "λt.λf.t"
```

In [18]:

```
eval @@ app [is_zero; one]
```

```
= (λs.λz.s z) (λy.λt.λf.f) (λt.λf.t)
= (λz.(λy.λt.λf.f) z) (λt.λf.t)
= (λy.λt.λf.f) (λt.λf.t)
= λt.λf.f
```

Out[18]:

```
- : string = "λt.λf.f"
```

Arithmetic

```
plus = λm.λn.λs.λz.m s (n s z)
mult = λm.λn.λs.m (n s)
```

In [19]:

```
let plus = p ("λm.λn.λs.λz.m s (n s z)")
let mult = p ("λm.λn.λs.m (n s)")
```

Out[19]:

```
val plus : Syntax.expr =
  Lam ("m",
   Lam ("n",
    Lam ("s",
     Lam ("z",
      App (App (Var "m", Var "s"), App (App (Var "n", Var
"s"), Var "z"))))))
```

Out[19]:

```
val mult : Syntax.expr =
  Lam ("m", Lam ("n", Lam ("s", App (Var "m", App (Var "n",
Var "s")))))
```

Arithmetic: addition

In [20]:

```
eval @@ app [plus; one; two]
```

```
= (λn.λs.λz.(λs.λz.s z) s (n s z)) (λs.λz.s (s z))
= λs.λz.(λs.λz.s z) s ((λs.λz.s (s z)) s z)
= λs.λz.(λz.s z) ((λs.λz.s (s z)) s z)
= λs.λz.s ((λs.λz.s (s z)) s z)
= λs.λz.s ((λz.s (s z)) z)
= λs.λz.s (s (s z))
```

Out[20]:

```
- : string = "λs.λz.s (s (s z))"
```

Proves 1 + 2 = 3. Can build a theory of arithmetic over lambda calculus.

Arithmetic: multiplication

In [21]:

```
eval @@ app [mult; three; two]
```

```
= (λn.λs.(λs.λz.s (s (s z))) (n s)) (λs.λz.s (s z))
```

Out[21]:

```
- : string = "λs.λz.s (s (s (s (s (s z)))))"
```

Arithmetic: predecessor

It turns out predecessor function is much more tricky compared to successor.

```
zz = pair zero zero
ss = λp. pair (snd p) (plus one (snd p))
```

```
zz = (0,0)
ss zz = (0,1)
ss (ss zz) = (1,2)
ss (ss (ss zz)) = (2,3)
```

etc.

Arithmetic: predecessor

It turns out predecessor function is much more tricky compared to successor.

```
zz = pair zero zero
ss = λp. pair (snd p) (plus one (snd p))
prd = λm. fst (m ss zz)
```

In [22]:

```
let zz = app [pair; zero; zero]
let ss = lam "p" (app [pair; app [snd; var "p"]; app [plus; one; app [sn
let prd = lam "m" (app [fst; app [var "m"; ss; zz]])
```

```
= λs.(λs.λz.s (s (s z))) ((λs.λz.s (s z)) s)
= λs.λz.(λs.λz.s (s z)) s ((λs.λz.s (s z)) s ((λs.λz.s (s
z)) s z))
= λs.λz.(λz.s (s z)) ((λs.λz.s (s z)) s ((λs.λz.s (s z)) s
z))
= λs.λz.s (s ((λs.λz.s (s z)) s ((λs.λz.s (s z)) s z)))
= λs.λz.s (s ((λz.s (s z)) ((λs.λz.s (s z)) s z)))
= λs.λz.s (s (s (s ((λs.λz.s (s z)) s z))))
= λs.λz.s (s (s (s ((λz.s (s z)) z))))
= λs.λz.s (s (s (s (s (s z)))))
```

Out[22]:

```
val zz : Syntax.expr =
  App
   (App
     (Lam ("f", Lam ("s", Lam ("b", App (App (Var "b", Var
"f"), Var "s")))),
     Lam ("s", Lam ("z", Var "z"))),
   Lam ("s", Lam ("z", Var "z")))
```

Out[22]:

```
val ss : Syntax.expr =
  Lam ("p",
   App
    (App
      (Lam ("f", Lam ("s", Lam ("b", App (App (Var "b", Var
"f"), Var "s")))),
      App (Lam ("p", App (Var "p", Lam ("t", Lam ("f", Var
"f")))), Var "p")),
    App
     (App
       (Lam ("m",
         Lam ("n",
          Lam ("s",
           Lam ("z",
            App (App (Var "m", Var "s"),
             App (App (Var "n", Var "s"), Var "z")))))),
       Lam ("s", Lam ("z", App (Var "s", Var "z")))),
     App (Lam ("p", App (Var "p", Lam ("t", Lam ("f", Var
"f")))), Var "p"))))
```

Out[22]:

```
val prd : Syntax.expr =
```

```
  Lam ("m",
   App (Lam ("p", App (Var "p", Lam ("t", Lam ("f", Var
"t")))),
    App
     (App (Var "m",
       Lam ("p",
        App
         (App
           (Lam ("f",
             Lam ("s", Lam ("b", App (App (Var "b", Var
"f"), Var "s")))),
            App (Lam ("p", App (Var "p", Lam ("t", Lam ("f",
Var "f")))),
             Var "p")),
          App
           (App
             (Lam ("m",
               Lam ("n",
                Lam ("s",
                 Lam ("z",
                  App (App (Var "m", Var "s"),
                   App (App (Var "n", Var "s"), Var
"z")))))),
              Lam ("s", Lam ("z", App (Var "s", Var "z")))),
            App (Lam ("p", App (Var "p", Lam ("t", Lam ("f",
Var "f")))),
             Var "p"))))),
      App
       (App
         (Lam ("f",
           Lam ("s", Lam ("b", App (App (Var "b", Var "f"),
Var "s")))),
          Lam ("s", Lam ("z", Var "z"))),
        Lam ("s", Lam ("z", Var "z")))))
```

Arithmetic: Predecessor

In [23]:

```
eval ~log:false @@ app [prd; three]
```

Out[23]:

```
- : string = "λs.λz.s (s z)"
```

In [24]:

```
eval ~log:false @@ app [prd; zero]
```

Out[24]:

```
- : string = "λs.λz.z"
```

Arithmetic: Subtraction

`sub` computes `m-n` :

```
sub = λm.λn.n prd m
```

Intuition: apply predecessor `n` times on `m` .

In [25]:

```
let sub = lam "m" (lam "n" (app [var "n"; prd; var "m"]))
```

Out[25]:

```
val sub : Syntax.expr =
  Lam ("m",
   Lam ("n",
    App
     (App (Var "n",
       Lam ("m",
        App (Lam ("p", App (Var "p", Lam ("t", Lam ("f", Va
r "t")))),
         App
          (App (Var "m",
            Lam ("p",
             App
              (App
                (Lam ("f",
                  Lam ("s", Lam ("b", App (App (Var "b", Va
r "f"), Var "s")))),
                App (Lam ("p", App (Var "p", Lam ("t", Lam
("f", Var "f")))),
                 Var "p")),
              App
               (App
                 (Lam ("m",
                   Lam ("n",
                    Lam ("s",
                     Lam ("z",
                      App (App (Var "m", Var "s"),
                       App (App (Var "n", Var "s"), Var
"z")))))),
                 Lam ("s", Lam ("z", App (Var "s", Var
"z")))),
               App (Lam ("p", App (Var "p", Lam ("t", Lam
("f", Var "f")))),
                Var "p"))))),
          App
           (App
             (Lam ("f",
               Lam ("s", Lam ("b", App (App (Var "b", Var
"f"), Var "s")))),
             Lam ("s", Lam ("z", Var "z"))),
           Lam ("s", Lam ("z", Var "z"))))))),
     Var "m")))
```

Arithmetic: Subtraction

In [26]:

```
eval ~log:false @@ app [sub; three; two]
```

Out[26]:

```
- : string = "λs.λz.s z"
```

In [27]:

```
eval ~log:false @@ app [sub; two; three]
```

Out[27]:

```
- : string = "λs.λz.z"
```

Arithmetic: equal

- `m - n = 0` $\implies$ `m = n`.
  - But we operate on natural numbers.
  - `3 - 4 = 0` $\implies$ `3 = 4`.
- `m - n = 0 && n - m = 0` $\implies$ `m = n`.

In [28]:

```
let equal =
  let mnz = app [is_zero; app [sub; var "m"; var "n"]] in
  let nmz = app [is_zero; app [sub; var "n"; var "m"]] in
  lam "m" (lam "n" (app [and_; mnz; nmz]))
```

Out[28]:

```
val equal : Syntax.expr =
  Lam ("m",
   Lam ("n",
    App
     (App
       (Lam ("b",
         Lam ("c",
          App (App (Var "b", Var "c"), Lam ("t", Lam ("f",
Var "f"))))),
       App
        (Lam ("n",
          App (App (Var "n", Lam ("y", Lam ("t", Lam ("f",
Var "f")))),
           Lam ("t", Lam ("f", Var "t")))),
        App
         (App
           (Lam ("m",
             Lam ("n",
              App
               (App (Var "n",
                 Lam ("m",
                  App
                   (Lam ("p", App (Var "p", Lam ("t", Lam
("f", Var "t")))),
                   App
                    (App (Var "m",
                      Lam ("p",
                       App
                        (App
                          (Lam ("f",
                            Lam ("s",
                             Lam ("b", App (App (Var "b", V
ar "f"), Var "s")))),
                          App
                           (Lam ("p",
                             App (Var "p", Lam ("t", Lam
("f", Var "f")))),
                           Var "p")),
                        App
                         (App
                           (Lam ("m",
```

```
                             Lam ("n",
                              Lam ("s",
                               Lam ("z",
                                App (App (Var "m", Var
"s"),
                                 App (App (Var "n", Var
"s"), Var "z")))))),
                           Lam ("s", Lam ("z", App (Var
"s", Var "z")))),
                         App
                          (Lam ("p",
                            App (Var "p", Lam ("t", Lam
("f", Var "f")))),
                          Var "p"))))),
                    App
                     (App
                       (Lam ("f",
                         Lam ("s",
                          Lam ("b", App (App (Var "b", Var
"f"), Var "s")))),
                       Lam ("s", Lam ("z", Var "z"))),
                     Lam ("s", Lam ("z", Var "z"))))))),
               Var "m"))),
           Var "m"),
         Var "n"))),
     App
      (Lam ("n",
        App (App (Var "n", Lam ("y", Lam ("t", Lam ("f", Va
r "f")))),
         Lam ("t", Lam ("f", Var "t")))),
      App
       (App
         (Lam ("m",
           Lam ("n",
            App
             (App (Var "n",
               Lam ("m",
                App (Lam ("p", App (Var "p", Lam ("t", Lam
("f", Var "t")))),
                 App
                  (App (Var "m",
                    Lam ("p",
                     App
                      (App
                        (Lam ("f",
                          Lam ("s",
                           Lam ("b", App (App (Var "b", Var
"f"), Var "s")))),
                        App
                         (Lam ("p",
```

```
                              App (Var "p", Lam ("t", Lam
("f", Var "f")))),
                            Var "p")),
                         App
                          (App
                            (Lam ("m",
                              Lam ("n",
                               Lam ("s",
                                Lam ("z",
                                 App (App (Var "m", Var "s"),
                                  App (App (Var "n", Var "s"),
Var "z")))))),
                            Lam ("s", Lam ("z", App (Var "s",
Var "z")))),
                          App (Lam (...), ...))))),
                     ...)))),
                ...))),
            ...),
          ...)))))
```

Arithmetic: equal

In [29]:

```
eval ~log:false @@ app [equal; two; two]
```

Out[29]:

```
- : string = "λt.λf.t"
```

In [30]:

```
eval ~log:false @@ app [equal; app[sub; three; two]; two]
```

Out[30]:

```
- : string = "λt.λf.f"
```

In [31]:

```
eval ~log:false @@ app [equal; app[sub; two; three]; zero]
```

Out[31]:

```
- : string = "λt.λf.t"
```

Fixed points

- Given a function f, if $x = f(x)$ then x is said to be a fixed point for f.
  - $f(x) = x^2$ has two fixed points 0 and 1.
  - $f(x) = x + 1$ has no fixed points.
- For lambda calculus, N is said to be a fixed point of F if $F\ N =_\beta N$
  - In the untyped lambda calculus, every term F has a fixed point!

Fixed points

- Let `D = λx.x x`, then
  - `D D = (λx.x x) (λx.x x)` $\rightarrow_\beta$ `(λx.x x) (λx.x x) = D D.`
- So `D D` is an infinite loop
  - In general, self-application is how you get looping

Fixed points

Let $Y = \lambda f.(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$, then

$$
\begin{aligned}
& Y\ F = (\lambda f.(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ F \\
\rightarrow_\beta\quad & (\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x)) \\
\rightarrow_\beta\quad & F((\lambda x.\ F(xx))(\lambda x.\ F(xx))) \\
\rightarrow_\beta\quad & F\ (Y\ F)
\end{aligned}
$$

- Therefore, `Y F = F(Y F).`
  - `Y F` is said to be the fixed point of `F`.
  - `Y F = F(Y F) = F(F(Y F)) = ...`
  - `Y` (`y`-combinator) can be used to achieve recursion.

Fixed point: Factorial

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

- Second argument `n` is the integer.
- First argument `f` is the function to call for the recursive case.
- We will use y-combinator to achieve recursion.

Fixed point: Factorial

$$
\begin{aligned}
& (Y \text{ fact})\ 1 = \ \text{fact}\ (Y \text{ fact})1 \\
\rightarrow_{beta} \quad & \text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * ((Y \text{ fact})\ 0) \\
\rightarrow_{beta} \quad & 1 * ((Y \text{ fact})\ 0) \\
\rightarrow_{beta} \quad & 1 * (\text{fact}\ (Y \text{ fact})\ 0) \\
\rightarrow_{beta} \quad & 1 * \text{if } 0 = 0 \text{ then } 1 \text{ else } 1 * ((Y \text{ fact})\ 0) \\
\rightarrow_{beta} \quad & 1 * 1 \\
\rightarrow_{beta} \quad & 1
\end{aligned}
$$

Fixed point: Factorial

In [32]:

```
let y = p "λf.(λx.f (x x)) (λx.f (x x))"
let fact =
  let tst = app [is_zero; var "n"] in
  let fb = app [mult; var "n"; app [var "f"; app [prd; var "n"]]] in
  lam "f" (lam "n" (app [tst; one; fb]))
```

Out[32]:

```
val y : Syntax.expr =
  Lam ("f",
   App (Lam ("x", App (Var "f", App (Var "x", Var "x"))),
    Lam ("x", App (Var "f", App (Var "x", Var "x")))))
```

Out[32]:

```
val fact : Syntax.expr =
  Lam ("f",
   Lam ("n",
    App
     (App
       (App
         (Lam ("n",
           App (App (Var "n", Lam ("y", Lam ("t", Lam ("f",
Var "f")))),
            Lam ("t", Lam ("f", Var "t")))),
         Var "n"),
       Lam ("s", Lam ("z", App (Var "s", Var "z")))),
      App
       (App
         (Lam ("m",
           Lam ("n", Lam ("s", App (Var "m", App (Var "n", V
ar "s"))))),
         Var "n"),
       App (Var "f",
        App
         (Lam ("m",
           App (Lam ("p", App (Var "p", Lam ("t", Lam ("f",
Var "t")))),
            App
             (App (Var "m",
               Lam ("p",
                App
                 (App
                   (Lam ("f",
                     Lam ("s",
                      Lam ("b", App (App (Var "b", Var "f"),
Var "s")))),
                   App
                    (Lam ("p", App (Var "p", Lam ("t", Lam
```

```
("f", Var "f")))),
                   Var "p")),
                App
                 (App
                   (Lam ("m",
                     Lam ("n",
                      Lam ("s",
                       Lam ("z",
                        App (App (Var "m", Var "s"),
                         App (App (Var "n", Var "s"), Var
"z")))))),
                   Lam ("s", Lam ("z", App (Var "s", Var
"z")))),
                 App
                  (Lam ("p", App (Var "p", Lam ("t", Lam
("f", Var "f")))),
                  Var "p"))))),
            App
             (App
               (Lam ("f",
                 Lam ("s", Lam ("b", App (App (Var "b", Var
"f"), Var "s")))),
               Lam ("s", Lam ("z", Var "z"))),
             Lam ("s", Lam ("z", Var "z")))))),
        Var "n"))))))
```

In [33]:

```
eval ~log:false @@ app [y; fact; one]
```

Out[33]:

```
- : string = "λs.λz.s z"
```

Quiz

The y-combinator Y = λf.(λx.f (x x)) (λx.f (x x)) is a fixed pointer combinator under which reduction strategy?

1. Call-by-value
2. Call-by-name
3. Both
4. Neither

Quiz

The y-combinator Y = λf.(λx.f (x x)) (λx.f (x x)) is a fixed pointer combinator under which reduction strategy?

1. Call-by-value
2. Call-by-name ✅
3. Both
4. Neither

Under call-by-value, we will keep indefinitely expanding `Y F = F (Y F) = F (F (Y F)) = ...`.

Fixed point: Z combinator

There is indeed a fixed point combinator for call-by-value called the Z combinator

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

which is just an η-expansion of the Y combinator

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

Fixed point: Z combinator

$$
\begin{aligned}
& Z\ F = (\lambda\ f\ .(\lambda x. f\ (\lambda y. x\ x\ y))\ (\lambda x. f\ (\lambda y. x\ x\ y)))\ \ F \\
\rightarrow_{\beta V} \quad & (\lambda\ x\ . F\ (\lambda y. x\ x\ y))\ \ (\lambda x. F\ (\lambda y. x\ x\ y)) \\
\rightarrow_{\beta V} \quad & F\ (\lambda y.\ \ (\lambda x. F\ (\lambda y. x\ x\ y))\ (\lambda x. F\ (\lambda y. x\ x\ y))\ \ y) \\
\rightarrow_{\beta V} \quad & F\ (\lambda y. (Z\ \ F)\ y)
\end{aligned}
$$

The η-expansion has prevented further reduction.

Fin.