λ-CALCULUS SYNTAX

```
e ::= v          variable reference
    | λv.e       lambda expression / abstraction
    | (e e)      procedure calls
```

λ-CALCULUS SEMANTICS

$$(\lambda x.E\ M) \longrightarrow E\{M/x\}$$

where we choose a "fresh x", by α-renaming the lambda-expression if necessary.

A "Scheme-like" representation of λv.e is:

```
(lambda (v) e)
```

Normal vs Applicative Order

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

In normal order (call-by-name)

$$(\lambda x.\underbrace{y}_{E}\ \underbrace{(\lambda x.(x\ x)\ \lambda x.(x\ x))}_{M})$$

$\xrightarrow{\beta}\ y$

In applicative order (call-by-value)

$$(\lambda x.y\ (\lambda x.\underbrace{(x\ x)}_{E}\ \underbrace{\lambda x.(x\ x)}_{M}))$$

$\xrightarrow{\beta}\ (\lambda x.y\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))$

α-RENAMING

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

$\xrightarrow{\alpha}$ $(\lambda z.(y\ z)\ x)$

Only <u>bound</u> variables can be renamed. No <u>free</u> variables can be captured (become bound) in the process. For example, we <u>cannot</u> α-rename x to y.

β-REDUCTION

$$(\lambda x.E\ M) \xrightarrow{\beta} E\{M/x\}$$

May require α-renaming to prevent capturing free variable occurrences.

For example:

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

$$\xrightarrow{\alpha} (\lambda x.\lambda z.(x\ z)\ (y\ w))$$

$$\xrightarrow{\beta} \lambda z.((y\ w)\ z)$$

η-CONVERSION

$$\lambda x.(E\ x) \xrightarrow{\eta} E$$

if x is <u>not</u> free in E.

For example:

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

$$\xrightarrow{\alpha} (\lambda x.\lambda z.(x\ z)\ (y\ w))$$

$$\xrightarrow{\beta} \lambda z.((y\ w)\ z)$$

$$\xrightarrow{\eta} (y\ w)$$

Combinators

A λ-calculus expression with no free variables is called a combinator. For example:

Identity $I = \lambda x.x$

Application $App = \lambda f.\lambda x.(f\ x)$

Currying $Cur = \lambda f.\lambda x.\lambda y.((f\ x)\ y)$

Sequencing $Seq = \lambda x.\lambda y.(\lambda z.y\ x)$ (normal order)

$ASeq = \lambda x.\lambda y.(y\ x)$ (applicative order)

where 'y' is a lambda abstraction 'wrapping' the second expression to evaluate.

Loop $Loop = (\lambda x.(x\ x)\ \lambda x.(x\ x))$

SEQUENCING COMBINATOR

$Seq = \lambda x.\lambda y.(\lambda z.y\ x)$

(Normal Order)

$ASeq = \lambda x.\lambda y.(y\ x)$

(Applicative Order)

with the 2nd. argument as a thunk.

Recursion (Y) Combinator

$$n! = \begin{cases} 1 & , n=0 \\ n*(n-1)! & , n>0 \end{cases}$$

```
f: λg.λn.(if (= n 0)
             1
             (* n (g (- n 1))))
```

$f : (\tau \to \tau) \to (\tau \to \tau)$

$(f\ f)$ ✗ Not well-typed.

$(f\ I)$ $\quad f(n) = \begin{cases} 1 & , n=0 \\ n*(n-1) & , n>0 \end{cases}$

$(f\ x) = X$

X: (λx.(λg.λn.(if (= n 0)
⊥
(* n (g (- n
))

λy.((x x) y)))

λx.(λg.λn.(if (= n 0)
1
(* n (g (- n 0)))
))

X can be defined as (Y F)
where Y is the recursion
combinator.

$$(f\ X) = \underline{(f\ (Y\ F)) = (Y\ F)}$$

Exercise: Prove equation
holds, for a
given f.

Recursion Combinator (Y)

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

(normal order)

$$Y = Rec \equiv \lambda f.(\lambda x.(f\ \lambda y.((x\ x)\ y))\ \lambda x.(f\ \lambda y.((x\ x)\ y)))$$

(applicative order)

You get from the normal order to the applicative order recursion combinator by η-expansion (η-conversion from right to left).

NATURAL NUMBERS IN λ-CALCULUS

$$[\![0]\!] = \lambda x.x$$

$$[\![1]\!] = \lambda x.\lambda x.x$$

$$\vdots$$

$$[\![n+1]\!] = \lambda x.[\![n]\!]$$

BRANCHING (IF) IN λ-CALCULUS (NORMAL ORDER)

true $\lambda x.\lambda y.x$ $V \to (V \to V)$

false $\lambda x.\lambda y.y$ $V \to (V \to V)$

if $\lambda b.\lambda t.\lambda e.((b\ t)\ e)$

$(V \to (V \to V)) \to E \to (E \to E)$