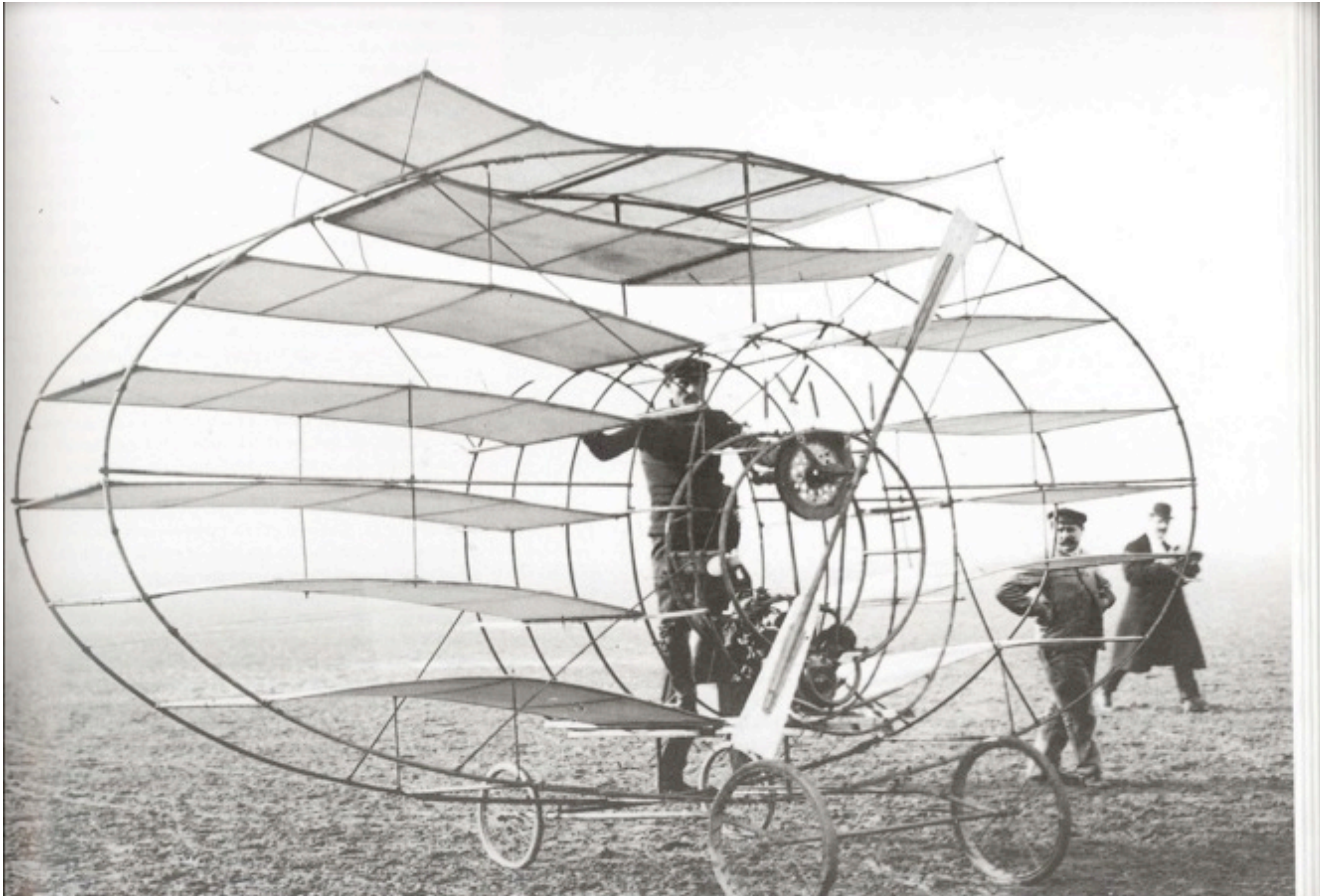
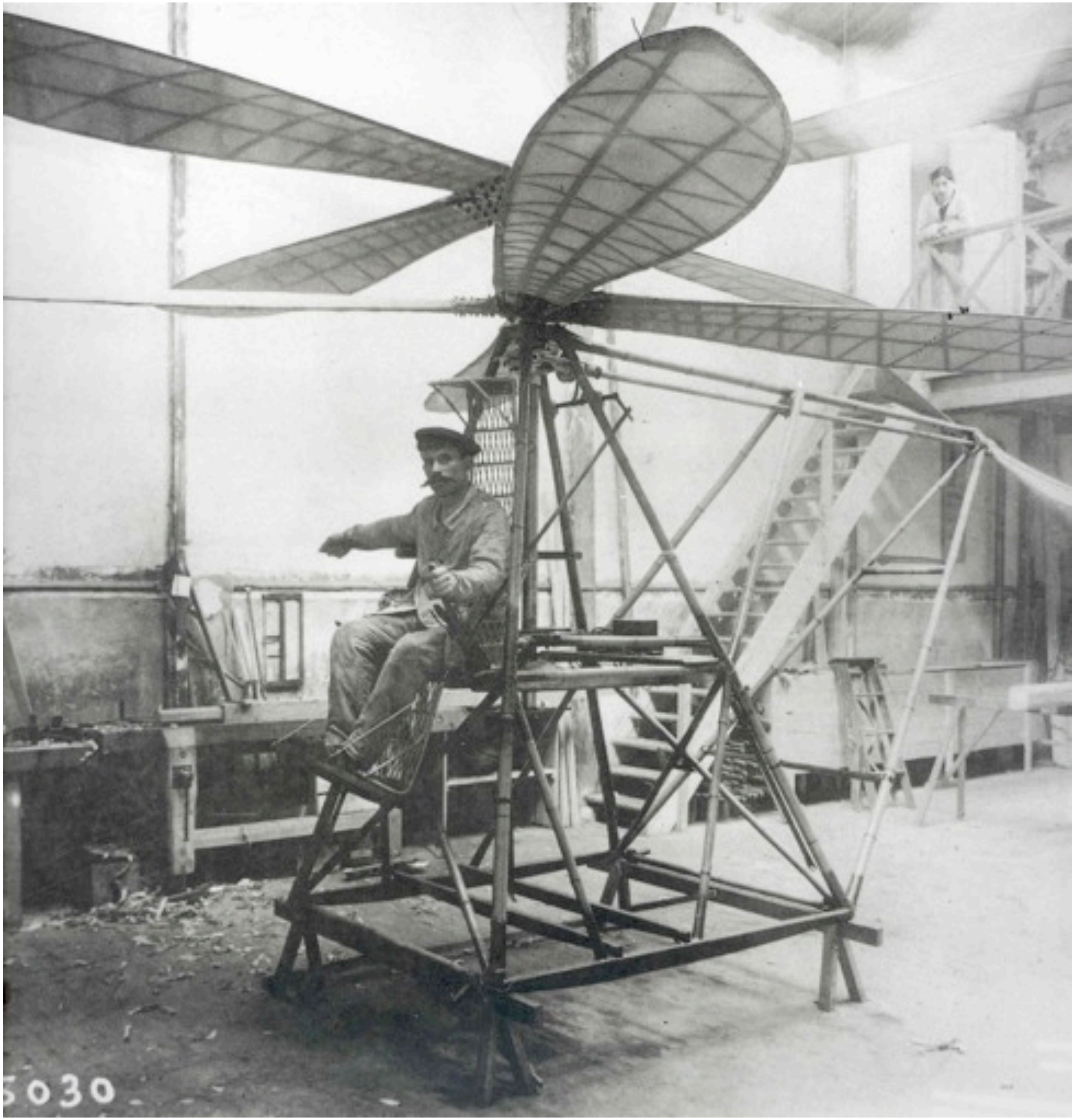



5030

프로그래밍 언어를 제대로 디자인해보자

IV. λ- Calculus

To learn :
- minimal language for general-purpose programming
- semantic variations : reduction strategies
- basis for higher-order & typed programming

Def. A function is computable iff
- [Gödel] it is recursive.
- [Turing] it runs on a Turing machine.
- [Church] it is definable in λ-Calculus.

- Above three definitions are proved to be equivalent. Hence we "believe" that computable functions are no more than that.
- "λ-Calculus as the programming language" is a good idea.

Syntax of λ-Terms

$$
\begin{aligned}
e \rightarrow\ & x && \text{variable}\\
\mid\ & \lambda x . e && \text{abstraction}\\
\mid\ & e\ e && \text{application}
\end{aligned}
$$

- unamed function $\lambda x . e$ binds x in e.

$\underset{\text{binder}}{\lambda}\ \underset{\text{bound variable}}{x}\ .\ \underset{\text{scope}}{e}$

- application is left-associative:
  - "$e_1\ e_2\ e_3$" is read (parsed) $((e_1\ e_2)\ e_3)$
- if shift-reduce conflict, then shift!
  - "$\lambda x.\ x\ \lambda y.\ y\ x$" is read (parsed) $\lambda x.(x(\lambda y.(y\ x)))$

- $\lambda x. x+1$ increment function $\in \mathbb{N} \to \mathbb{N}$
- $\lambda(x,y). x+y$ addition function $\in \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
- $\lambda x. \lambda y. x+y$ curried addition fn $\in \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$

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

- $(\lambda x. x+1)\, 2 \to 2+1 \to 3$
- $(\lambda(x,y). x+y)\,(1,2) \to 1+2 \to 3$
- $((\lambda x. (\lambda y. (x+y)))\, 1)\, 2 \to (\lambda y. 1+y)\, 2$

  $\to 1+2 \to 3$

- $(\lambda x. x\, 1)$ fn that applies a fn $\in (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$
- $(\lambda x (\lambda y. x\, y)) \in (A \to B) \to (A \to B)$
- $(\lambda x. x\, 1)\, (\lambda x. x+1) \to (\lambda x. x+1)\, 1$

  $\to 1+1 \to 2$

- $(\lambda x. (\lambda y. x\, y))\, (\lambda x. x+1)\, 2$

  $\to (\lambda y. (\lambda x. x+1)\, y)\, 2 \to \dots$

{y/x, z/y} x y
~~= z z~~
= y z

{y/x} \x.x
~~= \x.y~~
= \z.{y/x}{z/x}x
= \z.z

{x/y} \x.y
~~= \x.x~~
= \z.{x/y}{z/x}y
= \z.x

Transition Semantics of λ-Terms

(notations)

- Free variables

$FV(x) = \{x\}$

$FV(\lambda x.e) = FV(e) \setminus \{x\}$

$FV(e_1\, e_2) = FV(e_1) \cup FV(e_2)$

- Substitution $S = \{e_1/x_1, \cdots, e_n/x_n\}$

substitute λ-term e_i for variable x_i.

$$S\,x = \begin{cases} e & \text{if } e/x \in S \\ x & \text{o.w.} \end{cases}$$

$S(\lambda x.e) = \lambda x'.\, S\{x'/x\}\, e$

where $x' \notin \cup\{FV(e) \mid e/x \in S\}$
$\cup\, Supp\, S$
$\cup\, FV(e) \setminus \{x\}$

$S(e_1\, e_2) = (S\, e_1)(S\, e_2)$

이미 묶여있는 놈은 건들지 않는다.
새롭게 묶이는 놈이 없도록 한다.

[] x

\x.(x [])

(\x.x y)(y [])

~~(\x.[])[]~~

Transition Semantics of λ-Terms

(notations)

- **Free variables**

$FV(x) = \{x\}$

$FV(\lambda x.e) = FV(e) \setminus \{x\}$

$FV(e_1\, e_2) = FV(e_1) \cup FV(e_2)$

- **Substitution** $S = \{e_1/x, \cdots, e_n/x_n\}$

substitute λ-term e_i for variable x_i.

$$S\,x = \begin{cases} e & \text{if } e/x \in S \\ x & \text{o.w.} \end{cases}$$

$S(\lambda x.e) = \lambda x'.\, S\{x'/x\}\, e$

where $x' \notin \cup\{FV(e) \mid e/x \in S\} \cup \mathrm{Supp}\, S \cup FV(e)\setminus\{x\}$

$S(e_1\, e_2) = (S\, e_1)(S\, e_2)$

- **Context** : λ-term with a hole []

$C \to [\,] \mid C\, e \mid e\, C \mid \lambda x.C$

C[]   C[e]

베타계산
함수적용

Transition Semantics of λ-Terms

(the transition relation → between terms)

- $(\lambda x.e_1)\, e_2 \xrightarrow[\beta]{} \{e_2/x\}\, e_1$ (β-reduction)

- $$\frac{e_1 \to e_2}{C[e_1] \to C[e_2]}$$

- $$\frac{x' \notin FV(\lambda x.e)}{\lambda x.e \xrightarrow[\alpha]{} \lambda x'.\{x'/x\}e}$$ (α-conversion)

* the β-reduction is the only way of doing computation. (computing is applying functions!)

* expression $(\lambda x.e_1)\, e_2$, which fires the β-reduction, is called **redex**.
(reduceable expression)

* a term without a redex is called **normal**.
(every computation is done, i.e., a value!)

Examples

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

$(\lambda x.(\lambda y.y\,x)\,z)\,(z\,w) \to$

$(\lambda x.(\lambda y.y\,x)\,z)\,(z\,w) \to$

$(\lambda x.x\,x)(\lambda x.x\,x) \to$

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

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

Examples

$(\lambda x.y)(\lambda z.z) \to$ `y`

$(\lambda x.(\lambda y.yx)z)(zw) \to$ `(\y.y(zw))z`

$\to$ `z(zw)`

$(\lambda x.(\lambda y.yx)z)(zw) \to$ `(\x.zx)(zw)`

$\to$ `z(zw)`

$(\lambda x.xx)(\lambda x.xx) \to$ `(\x.xx)(\x.xx)`

$(\lambda x.y)((\lambda x.xx)(\lambda x.xx)) \to$ `y`

$(\lambda x.y)((\lambda x.xx)(\lambda x.xx)) \to$ `(\x.y)((\x.xx)(\x.xx))`

* λ-term e's semantics as the transition (reduction) $\rightarrow$ sequence is not unique.

* λ-term e's semantics as its value (normal form, the last one in the $\rightarrow$ sequence) is "unique," because

<u>Thm</u> [Church-Rosser] [Confluence Theorem]

If $e \overset{*}{\nearrow} e_1$, $e \overset{*}{\searrow} e_2$ then $\exists e_3$. $e_1 \overset{*}{\searrow} e_3$, $e_2 \overset{*}{\nearrow} e_3$.

<u>Cor</u> If we regard e_1 & e_2 are equiv. whenever $e_1 \xrightarrow[\alpha]{} e_2$ then every term has at most one normal form.

dynamic scoping is a non-sense.

- Thus renaming bound-variables (α-conversions) must be meaningless.
- If the scoping is dynamic, then the renaming becomes meaningful.

e.g.) $(\lambda x.\ (\lambda x.\ x\ 1)\ (\lambda y. x+y))\ 1$ versus

$(\lambda x.\ (\lambda z.\ z\ 1)\ (\lambda y. x+y))\ 1$

* λ-term does <u>not</u> always have a normal form.

$$(\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \rightarrow$$

$$(\lambda x.\, f\, (x\, x))\, (\lambda x.\, f\, (x\, x)) \rightarrow$$

(non-terminating programs!)

* for λ-term <u>with</u> a normal form the reduction $\rightarrow$ rule does <u>not</u> guarantee to reach the normal form.

<u>Thm</u> [Standardization Theorem]
If a λ-term has a normal form then the <u>normal-order</u> reduction arrives at the normal form.

* Normal-order reduction

$$\frac{e_1 \rightarrow e_2}{C[e_1] \rightarrow C[e_2]}$$ e_1 is the <u>left-most</u> redex of the <u>outer-most</u> redius of $C[e_1]$.

eg. $((\lambda x.\, e_1)\, e_2)\, ((\lambda x. x)\, y)$

Programming in λ-Calculus

(with normal-order reduction)

Encodings of $\mathbb{N}$, $\mathbb{B}$, functions, branches, recursions.

Church numerals

$\underline{0} \triangleq \lambda f. \lambda x. x$

$\underline{1} \triangleq \lambda f. \lambda x. f\, x$

$\underline{n} \triangleq \lambda f. \lambda x. f^n x$

$\underline{\text{true}} \triangleq \lambda x. \lambda y. x$

$\underline{\text{false}} \triangleq \lambda x. \lambda y. y$

$\underline{\text{if } e_1\ e_2\ e_3} \triangleq \underline{e_1}\ \underline{e_2}\ \underline{e_3}$

$\underline{\text{not}} \triangleq \lambda b. \lambda x. \lambda y.\ b\, y\, x$

$\underline{\text{and}} \triangleq \lambda b. \lambda b'. \lambda x. \lambda y.\ b\,(c\, x\, y)\, y$

$\underline{\text{iszero}} \triangleq \lambda n. \lambda x. \lambda y.\ n\, (\lambda z. y)\, x$

$\underline{\text{succ}} \triangleq \lambda n. \lambda f. \lambda x. f\, (n\, f\, x)$

$\underline{\text{add}} \triangleq \lambda n. \lambda n'. \lambda f. \lambda x.\ n\, f\, (n'\, f\, x)$

$\underline{\text{mult}} \triangleq \lambda n. \lambda n'. \lambda f.\ n\, (n'\, f)$

- double
  \m.\s.\z.m s (m s z)
- not
  \b.b fls tru
- and
  \b.\c.b c fls
- add
  \m.\n.\s.\z.m s (n s z)
- mult
  \m.\n.m (add n) 0
- iszero?
  \m.m (\n.fls) tru
- pred
  \m. fst (m ss zz)
  where
  zz = pair 0 0

1+2

add 1 2

(\n\m\f\x.nf(mfx)) (\f\x.fx) (\f\x.f(fx))

→ (\m\f\x.(\f\x.fx)f(mfx)) (\f\x.f(fx))

→ \f\x.(\f\x.fx)f((\f\x.f(fx))fx)

→ \f\x.(\x.fx)((\f\x.f(fx))fx)

→ \f\x.f((\f\x.f(fx))fx)

→ \f\x.f((\x.f(fx))x)

→ \f\x.f(f(fx))

= 3

if false then 1 else 2

(\x\y.y) 1 2

→ (\y.y) 2

→ 2

Encoding of Recursive Functions in λ-Calculus

In ML, C, Java & etc.,

```
fun fac(n) = if n=0 then 1
             else n * fac(n-1)
```

In λ-Calculus,

$$\underline{fac} \triangleq Y(\lambda f. \lambda n. \ \underline{if}\ \underline{n=0}\ \underline{1}\ \underline{n \times f(n-1)})$$

$$Y \triangleq \lambda f. (\lambda x. f(x\,x))(\lambda x. f(x\,x))$$

the <u>fixpoint combinator</u>

c.f.) the denotational semantics $[\![fac]\!]$ is

$$[\![fac]\!] \triangleq \underline{fix}\ \lambda f. \lambda n. \text{ if } n=0 \text{ then } 1 \text{ else } n \times f(n-1)$$

c.f.) $Yf = f(Yf)$, i.e., Y forms a fixpoint of its argument function.

Then

$$\underline{fac}\ \underline{n} \xrightarrow{*} \underline{n!}$$

Y F 1

→ (\x.F(xx)) (\x.F(xx)) 1

GG1

→ F(GG)1

→ (\n.if n=0 1 n*((GG)(n–1))) 1

→ if 1=0 1 1*((GG)(1–1))

→ if false 1 1*((GG)(1–1))

→ 1*((GG)(1–1)) → 1*(F(GG)(1–1))

→ 1*(if (1–1)=0 1 1*((GG)((1–1)–1)))

→ 1*1

Examples

fac 1

$$= Y\ \underbrace{(\lambda f.\lambda n.\ \text{if } n=0\ 1\ n \times f(n-1))}_{F}\ 1$$

$$= (\lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x)))\ F\ 1$$

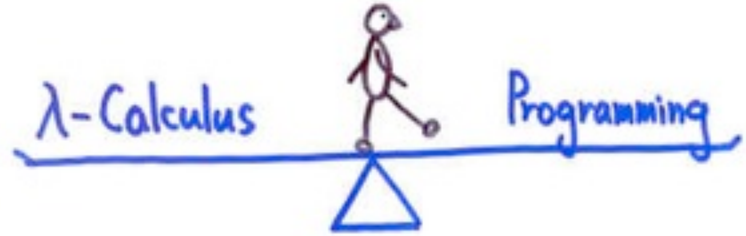
Did you see/feel the power of λ's ?

* all "computable" functions are encoded in pure λ-calculus.

* 놀랄만큼 간단한 계산 (β reduction) 방식이 위의 것을 가능하게 하는구나.

* $\lambda x.e$ 를 함수라고 생각하고 β-reduction을 함수의 적용이라고 읽는다면
함수가 자유자재로 정의되고 함수가 적용되면서
함수가, 전달되고 결과로 나오고, 하는 등의 일이
계산의 모든 것이구나.

"high-order computation"



Reduction 줄이기	$\longleftrightarrow$	Evaluation 계산하기
Normal form 더이상 줄일 수 없는	$\longleftrightarrow$	Canonical form 계산이 끝난 값
λ-term 대상	$\longleftrightarrow$	Closed λ-term 프로그램

- λ-term e is closed iff $FV(e) = \phi$
- λ-term e is canonical iff $e = \lambda x.e'$

We will use v for canonical forms.

Evaluation Strategies & Implementations

(normal-order evaluation
eager evaluation)

<u>Def</u> [$e \xrightarrow[N]{*}\!\!\!\!\rightarrow v$]

$e \xrightarrow[N]{*}\!\!\!\!\rightarrow v$ iff λ-expr e reduces into the first canonical form v in the normal-order reduction sequence.

<u>Evaluation Rules</u> (instead of reductions in λ-Calculus)

$$\frac{}{\lambda x.e \Rightarrow_N \lambda x.e}$$

$$\frac{e_1 \Rightarrow_N \lambda x.e' \qquad \{e_2/x\}e' \Rightarrow_N v}{e_1\ e_2 \Rightarrow_N v}$$

<u>Thm</u> $\forall$ closed expression e.

$e \xrightarrow[N]{*}\!\!\!\!\rightarrow v$ iff $e \Rightarrow_N v$.

pr) ($\Rightarrow$) By ind. on $\xrightarrow[N]{*}\!\!\!\!\rightarrow$ size ($\Leftarrow$) By ind. on $\overset{\nabla}{\overline{e \Rightarrow_N v}}$ size.

Def [$e \overset{*}{\underset{E}{\twoheadrightarrow}} v$]

$e \overset{*}{\underset{E}{\twoheadrightarrow}} v$ iff λ-expr e reduces into the first canonical form v in the eager-order reduction sequence.

Def [eager-order reduction rules]

- $(\lambda x.e)\, v \to \{v/x\}\, e$ (β_E-reduction)
- $\dfrac{e_1 \to e_2}{C[e_1] \to C[e_2]}$ e_1 is the left-most β_E-redex not inside a canonical form.

Evaluation Rules

$$\frac{}{\lambda x.e \underset{E}{\Rightarrow} \lambda x.e}$$

$$\frac{e_1 \underset{E}{\Rightarrow} \lambda x.e' \quad e_2 \underset{E}{\Rightarrow} v \quad \{v/x\}e' \underset{E}{\Rightarrow} v'}{e_1\, e_2 \underset{E}{\Rightarrow} v'}$$

Thm $\forall$ closed expression e.
$e \overset{*}{\underset{E}{\twoheadrightarrow}} v$ iff $e \underset{E}{\Rightarrow} v$.

- Normal-order evaluation is also called "lazy evaluation" or "call-by-name."
  소극적 계산법
- Eager evaluation is 적극적 계산법.
  or "call-by-value."

* Eager evaluation is not "normal."
The normal (canonical) form cannot be found sometimes.

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

* In eager evaluation we <u>cannot</u> encode <u>if $e_1\,e_2\,e_3$</u> and <u>recursive functions</u> the same as before.

eg) <u>if e_1 e_2 e_3</u> $\triangleq$ $\underline{e_1}$ $\underline{e_2}$ $\underline{e_3}$

$\underset{E}{\Rightarrow} v$ if (after) $e_2 \underset{E}{\Rightarrow} v_2 \wedge e_3 \underset{E}{\Rightarrow} v_3$

We need ifferent encoding to avoid the eager evaluation.

the semantic style that has a "right" gap from the implementation

Evaluation Rules

$$\boxed{\sigma \vdash e \Rightarrow v}$$

separation of syntactic objects, codes / semantic objects, values

$$\sigma \in Env = Var \xrightarrow{fin} Val$$

$$v \in Val = Lambda \times Env \qquad \text{"closures"}$$

- $$\frac{\sigma(x) = v}{\sigma \vdash x \Rightarrow v}$$

- $$\frac{}{\sigma \vdash \lambda x.e \Rightarrow \langle \lambda x.e, \sigma \rangle}$$

- $$\frac{\sigma \vdash e_1 \Rightarrow \langle \lambda x.e', \sigma' \rangle \quad \sigma \vdash e_2 \Rightarrow v \quad \sigma'[v/x] \vdash e' \Rightarrow v'}{\sigma \vdash e_1\, e_2 \Rightarrow v'}$$

Dynamic scoping 이 뭐이냐고?

- $$\frac{}{\sigma \vdash \lambda x.e \Rightarrow \lambda x.e}$$

- $$\frac{\sigma \vdash e_1 \Rightarrow \lambda x.e' \quad \sigma \vdash e_2 \Rightarrow v \quad \sigma[v/x] \vdash e' \Rightarrow v}{\sigma \vdash e_1 e_2 \Rightarrow v}$$

Thm [Correct Implementation]

$$\sigma \vdash e \Rightarrow v \quad \text{iff} \quad \underline{\sigma}\, e \underset{E}{\Rightarrow} \underline{v}$$

Def $\underline{\langle \lambda x.e, \sigma \rangle} \triangleq \underline{\sigma}(\lambda x.e)$

$$\underline{\phi} \triangleq \phi$$

$$\underline{\{v_1/x_1, \cdots, v_n/x_n\}} \triangleq \{\underline{v_1}/x_1, \cdots, \underline{v_n}/x_n\}$$

pf) ($\Rightarrow$) By induction on proof size of $\dfrac{\nabla}{\sigma \vdash e \Rightarrow v}$.

($\Leftarrow$) By induction on the length of $\underset{E}{\twoheadrightarrow}$ chain.

"두 방식이 같다."

Thus $\cdot \vdash \cdot \Rightarrow \cdot$ is a sound & complete implementation of $\underset{E}{\Rightarrow}$ (i.e. $\overset{*}{\underset{E}{\twoheadrightarrow}}$)

충실한 구현 = 안전하고 완전한.

V. Applicative Language

(an eager-evaluation language)

$e \rightarrow$	n	integer
$\mid$	x	variable
$\mid$	$\lambda x.e$	abstraction
$\mid$	$e\ e$	application
$\mid$	$\text{rec } f\ \lambda x.e$	recursive abstraction
$\mid$	$\text{ifo}\ e\ e\ e$	branch
$\mid$	$e + e$	primitive int. op.

Note

In eager-evaluation, "rec f λx.e" and "ifo ..." are still <u>syntactic sugar</u>; impossible to encode them using the usual λ-encoding (Y, etc) but possible with a variant of Y:

$$Z = \lambda f.(\lambda x.f(\lambda z.xx))(\lambda x.f(\lambda z.xx))$$

Natural Semantics

$$\boxed{\sigma \vdash e \Rightarrow v}$$

$$\sigma \in Env = Id \xrightarrow{fin} Val$$

$$v \in Val = (Expr \times Env) + Int$$

$$\sigma \vdash \lambda x.e \Rightarrow \langle \lambda x.e, \sigma \rangle$$

$$\sigma \vdash \text{rec } f\ \lambda x.e \Rightarrow \langle \text{rec} f\ \lambda x.e, \sigma \rangle$$

$$\frac{\begin{array}{l}\sigma \vdash e_1 \Rightarrow \langle \lambda x.e, \sigma' \rangle \\ \sigma \vdash e_2 \Rightarrow v \\ \sigma'[v/x] \vdash e \Rightarrow v'\end{array}}{\sigma \vdash e_1\, e_2 \Rightarrow v'}$$

$$\frac{\begin{array}{l}\sigma \vdash e_1 \Rightarrow \langle \text{rec} f\ \lambda x.e, \sigma' \rangle \\ \sigma \vdash e_2 \Rightarrow v \\ \sigma'[v/x][\langle \text{rec} f\ \lambda x.e, \sigma' \rangle / f] \\ \quad \vdash e \Rightarrow v'\end{array}}{\sigma \vdash e_1\, e_2 \Rightarrow v'}$$

$$\frac{}{\sigma \vdash x \Rightarrow v}\ v = \sigma(x)$$

And the usual rules for n, $e_1 + e_2$, ifo $e_1\ e_2\ e_3$.

Syntactic Sugars

달콤하지만 꼭 필요한 것은 아닌?

$$\boxed{\texttt{let } x = e \texttt{ in } e}$$

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma[v_1/x] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Rightarrow v_2}$$

$$\begin{aligned} &[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!] \\ &\quad \triangleq \lambda\sigma.(\lambda v.\, [\![e_2]\!]\sigma[v/x])_{\perp} \\ &\qquad\quad \varphi_{\perp}([\![e_1]\!]\sigma) \end{aligned}$$

$$(S, E, \texttt{let } x = e_1 \texttt{ in } e_2 . C, D) \to (S, E, e_1 . x^+ . e_2 . x^- . C, D)$$
$$(v.S, E, x^+ . C, D) \to (S, E[v/x], C, D)$$
$$(S, E, x^- . C, D) \to (S, E|_{dom(E)\setminus\{x\}}, C, D)$$

$$\texttt{let } x = e_1 \texttt{ in } e_2 \approx (\lambda x.e_2)\, e_1$$

Thm. $\sigma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Rightarrow v_2$ iff $\sigma \vdash (\lambda x.e_2)\, e_1 \Rightarrow v_2$

Thm. $[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!] = [\![(\lambda x.e_2)\, e_1]\!]$.

Thm. $(S, E, \texttt{let } x = e_1 \texttt{ in } e_2 . C, D) \xrightarrow{*} (v.S, E, C, D)$
iff $(S, E, (\lambda x.e_2)\, e_1 . C, D) \xrightarrow{*} (v.S, E, C, D)$.

$$\boxed{(e,e) \mid e.1 \mid e.2}$$

$Val = Int + Expr \times Env + Val \times Val$

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash (e_1, e_2) \Rightarrow \langle v_1, v_2 \rangle} \qquad \frac{\sigma \vdash e \Rightarrow \langle v_1, v_2 \rangle}{\sigma \vdash e.i \Rightarrow v_i}$$

- Encodings

$$\left[\begin{array}{lcl} \underline{(e_1, e_2)} & \triangleq & (\lambda l. \lambda r. \lambda f. f\ l\ r)\ \underline{e_1}\ \underline{e_2} \\ \underline{e.1} & \triangleq & \underline{e}\ (\lambda l. \lambda r. l) \\ \underline{e.2} & \triangleq & \underline{e}\ (\lambda l. \lambda r. r) \end{array}\right]$$

$$\begin{array}{lcl} \underline{x} & \triangleq & x \\ \underline{\lambda x. e} & \triangleq & \lambda x. \underline{e} \\ \underline{e_1\ e_2} & \triangleq & \underline{e_1}\ \underline{e_2} \\ \underline{\text{rec } f\ \lambda x. e} & \triangleq & \text{rec } f\ \lambda x. \underline{e} \end{array}$$

- Then $\quad e \approx \underline{e}$

Thm. $\sigma \vdash e \Rightarrow v$ iff $\underline{\sigma} \vdash \underline{e} \Rightarrow \underline{v}$

where

$$\begin{array}{rcl} \underline{\langle v_1, v_2 \rangle} & \triangleq & \lambda f. f\ \underline{v_1}\ \underline{v_2} \\ \underline{v} & \triangleq & v \\ \underline{\langle \lambda x. e, \sigma \rangle} & \triangleq & \langle \lambda x. \underline{e}, \underline{\sigma} \rangle \\ \underline{\sigma} & \triangleq & \{ x \mapsto \underline{\sigma x} \mid x \in Dom\, \sigma \} \end{array}$$

- let $x = e_1$ in e_2 for $(\lambda x. e_2)\ e_1$

- let $x = e_1$, $y = e_2$ in e_3 for   let $x = e_1$ in let $y = e_2$ in e_3

- let $(x, y) = e_1$ in e_2 for $\left(\lambda z.\ \text{let } x = z.1,\ y = z.2 \text{ in } e_2\right) e_1$

e.g.)
```
let
   fac = rec f λn. if0 n 1  n×f(n-1)
in
   fac 2
```

e.g.)
```
let
   x=1
   f = λy. x+y
in
  let
    x=2
  in
   f 10
```

e.g.)
```
(let
   (f,a)
   =(λn. λm. n+m, 10)
 in
   f a) 5
```

e.g.) $(\lambda(x,y).\ x+y)\ (1,2)$

Adding Imperative Features

$$\boxed{\texttt{ref } e \mid e := e \mid\ !e}$$

Natural Semantics

$$v \in Val = Int + Expr \times Env + Val \times Val$$
$$+ \; Loc$$
$$\ell \in Loc$$
$$s \in Store = Loc \xrightarrow{fin} Val$$
$$\sigma \in Env = Var \xrightarrow{fin} Val$$

$$\boxed{s, \sigma \vdash e \Rightarrow v, s'}$$

malloc!

$$\frac{s, \sigma \vdash e \Rightarrow v, s'}{s, \sigma \vdash \texttt{ref } e \Rightarrow \ell, s'[v/\ell]} \; \ell \notin dom(s')$$

$$\frac{s, \sigma \vdash e \Rightarrow \ell, s_1}{s, \sigma \vdash e \Rightarrow s_1(\ell)} \; \ell \in dom(s_1)$$

$$\frac{\begin{array}{c} s, \sigma \vdash e_1 \Rightarrow \ell, s_1 \\ s_1, \sigma \vdash e_2 \Rightarrow v, s_2 \end{array}}{s, \sigma \vdash e_1 := e_2 \Rightarrow v, s_2[v/\ell]}$$

Other cases are the same as before except that sub-expression's effects on store are accumulated. (e.g.)

$$\frac{\begin{array}{c} s, \sigma \vdash e_1 \Rightarrow \langle \lambda x.e, \sigma' \rangle, s_1 \\ s_1, \sigma \vdash e_2 \Rightarrow v, s_2 \\ s_2, \sigma'[v/x] \vdash e \Rightarrow v', s_3 \end{array}}{s, \sigma \vdash e_1 e_2 \Rightarrow v', s_3}$$

Semantics Using Evaluation Context

- Evaluation Contexts

$$
\begin{aligned}
C \rightarrow\ & [\,] \\
\mid\ & C\ e \\
\mid\ & v\ C \\
\mid\ & \text{ifo}\ C\ e\ e \\
\mid\ & C + e \\
\mid\ & v + C \\
v \rightarrow\ & n \\
\mid\ & \lambda x.e \mid \text{rec}\ f\ \lambda x.e \\
\mid\ & x
\end{aligned}
$$

- Reduction / Transition Rules

$$\frac{e \rightarrow e'}{C[e] \rightarrow C[e']}$$

$$(\lambda x.e)\ v \to \{v/x\}\,e$$

$$(\mathrm{rec}\ f\ \lambda x.e\ )\ v \to \{v/x,\ \mathrm{rec}\ f\ \lambda x.e/f\}\,e$$

$$\mathrm{ifo}\ 0\ e_1\ e_2 \to e_1$$

$$\frac{}{\mathrm{ifo}\ n\ e_1\ e_2 \to e_2}\ n \neq 0$$

$$\frac{}{n_1 + n_2 \to n}\ n = n_1 + n_2$$

e.g.) $\mathrm{ifo}\ (\lambda x.x+1)\,2\ \ (\lambda x.x)\,1\ \ (\mathrm{rec}\ f\ \lambda x.$
$\quad \mathrm{ifo}\ x\ 1$
$\quad\quad x + f(n-1)$
$)\,1$

$\to$

e.g.) $(\lambda x.(\lambda x.x\ 1)(\lambda y.x+y))\ 1$

$\to$