λ Calculus (1)

- Alonzo Church (1930s)
  - math formulation for fns

λ calculus is to functional prog as Turing machines are to imperative prog

- the two are equivalent

can design a T. M. to interpret λ calc.
can write a λ calc prog to interpret T.M.

λ calculus → Lisp ⇄ Scheme → ocaml
Lisp ⇄ ML ← Haskell

lambda abstraction

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

application $(\lambda x. + 1\ x)\ 2$

reduction rule $\Rightarrow (+\ 1\ 2) \Rightarrow 3$

curried
$(+\ 1\ x)$
$((+1)\ x)$

Syntax:

expr → constant
→ variable
→ (expr expr)
→ (λ variable . expr)

constants: 0, 1, *
variables: x, y

- λcalc has no concept of memory
  referentially transparent
  pure functional prog

λ Calculus (2)

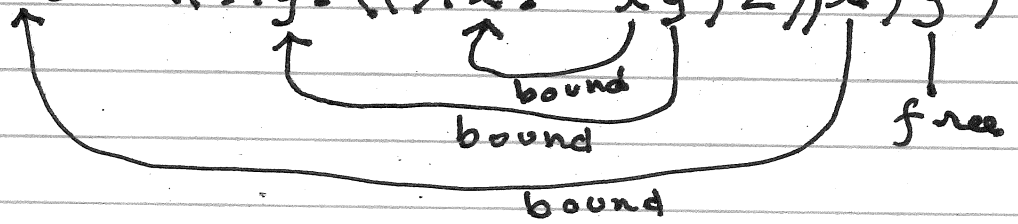
$(\lambda x.+1\,x)$ should be $(\lambda x.((+1)\,x))$ CURRIED

Var x in expr $(\lambda x.E)$ is <u>bound</u> by λ
- scope is E
- var outside scope is <u>free</u>

ex: $(\lambda x.+\,y\,x)$ ⟶ x is bound, y is free

diff vars bound by diff λ's.

$(\lambda x.+((\lambda y.((\lambda x.*\,x\,y)\,2))\,x)\,y)$



reduction of above:

$(\lambda x.+((\lambda y.((\lambda x.*\,x\,y)\,2))\,x)\,y)$
$\Rightarrow (\lambda x.+((\lambda y.(*\;2\;y))\,x)\,y)$
$\Rightarrow (\lambda x.+(*\;2\;x)\,y)$
$\Rightarrow (+\;(x\;2\;y))$

Juxtaposition of two exprs is <u>function application</u>.
substitute arg & eliminate λ

$$f\,g\,x \equiv (f\,g)\,x$$

$$f\,(g\,x)$$

β = beta

λCalculus ③

β reduction

$((\lambda x.+\ x\ 1)2) \Rightarrow (+\ 2\ 1)$

β abstraction

$(+\ 2\ 1) \Rightarrow (\lambda x.+\ x\ 1)\ 2$

β conversion refers to either & establishes equivalence

β conversion says:

$((\lambda x.E)F) \equiv E[F/x]$

where $E[F/x]$ is E with all free occurences of $x \in E$ replaced by F

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

bound free

Wrong $\not\Rightarrow (\lambda y.+\ y\ y)$ — can't

Name Capture: need to change the name of inner y

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

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

$\Rightarrow (\lambda z.+\ y\ z)$

α = alpha

λ calculus (4)

Namechange is α-conversion

$(\lambda x . E) \equiv (\lambda y . E[y/x])$

η = eta

η-conversion

-elimination of redundant λ abstractions

ex: $(\lambda x . (+\ 1\ x)) \equiv (\lambda x . (+1)\ x)$

1 is curried

(+1) is a fn of one argument

(+1) x is application of (+1) to x

thus

$(\lambda x . (+1)\ x) \equiv (+1)$

without the λ abstraction

$(\lambda x . (E\ x)) \equiv E$ by η-conversion

if E has no free x

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

$\Rightarrow +$

η-reduction useful in fn lang

~~Oscar dot sum fold~~

ex:

λ Calculus (5)

η-reduction useful in fn lang.

Ocaml: `fold_left ;;`

`('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

```
let sum list = fold_left (+) ∅ list ;;
let sum = fold_left (+) ∅ ;;
```

(example of Currying)

```
(+) ;;
int -> int -> int
let incr = (+) 1 ;; -> incr : int -> int
incr 3 ⟹ 4
```

Currying: partial function application

Evaluation order

applicative (pass by value)
- eager
- eval args then call

normal order (pass by name)
- lazy
- thunk (lambda())
- pass args are valuated
- delayed evaluation

example $((\lambda x.*\ x\ x)(+\ 2\ 3))$

applicative order

evalu. $\Rightarrow((\lambda x.*\ x\ x)\ 5)$

then β-red: $\Rightarrow (*\ 5\ 5)$

$\Rightarrow 25$

normal order

β-red 1st $\Rightarrow$

$\Rightarrow (*(+\ 2\ 3)(+\ 2\ 3))$

$\Rightarrow (*\ 5\ 5)$

$\Rightarrow 25$

~~Syntax trees~~:

possible applicative order undefined, but normal order returns value.

ex: $((\lambda x.xx)(\lambda x.xx))$

β reduction $\Rightarrow ((\lambda x.xx)(\lambda x.xx))$

↳ gives ∞ loop.

↳ elim λ and replace x with arg.

however

$((\lambda y.2)((\lambda x.xx)(\lambda x.xx)))$

normal order $\Rightarrow 2$

nonstrict: can return a value even if args undef

strict: always undef if arg undef

$\perp$ called bottom (undef)

applicative is strict

normal is non-strict

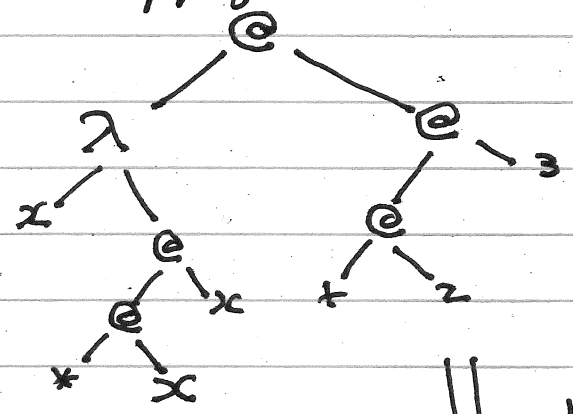
equivalence: reduce in normal order then compare.

graphical example
(syntax tree)

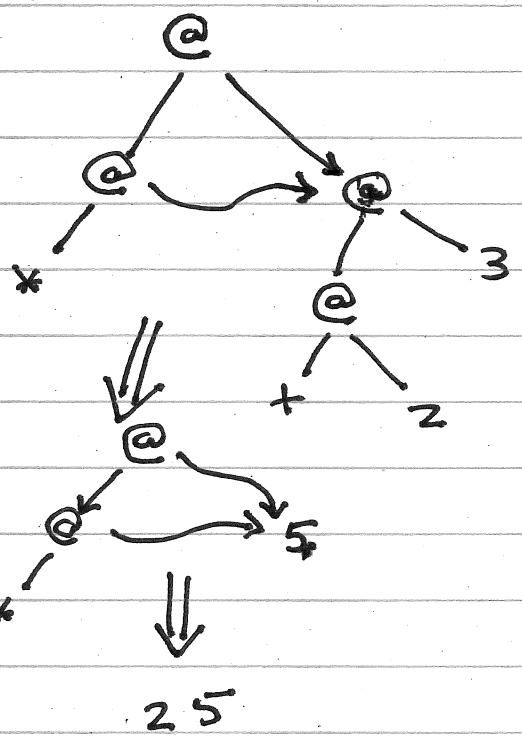
@ means apply

((λx.*xx)(+ 2 3))

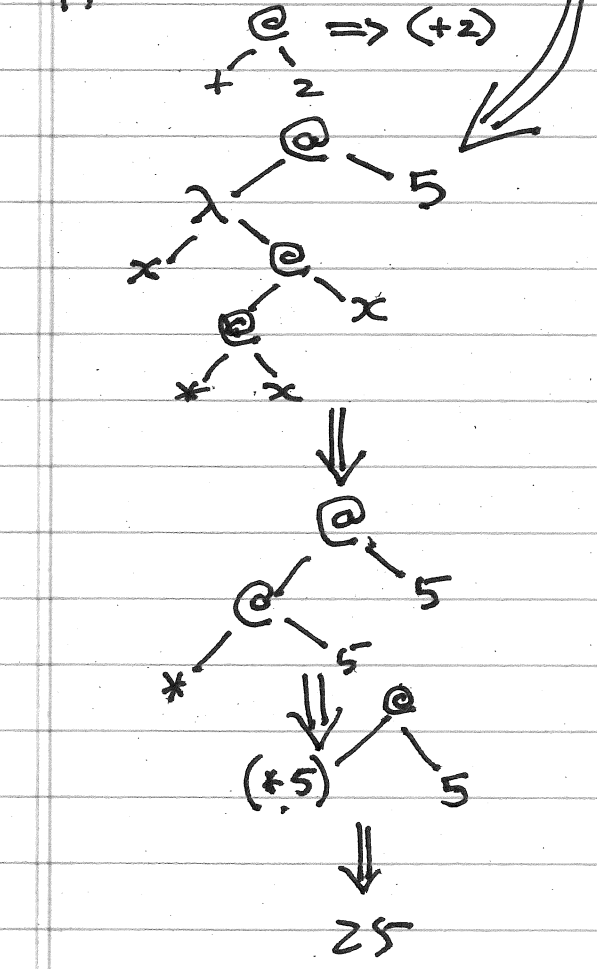
@ ≡ apply



normal order



applicative order



~~Factorial example~~

Recursive functions (factorial)

$$f = (\lambda n.\ (if\ (= n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))))$$

remove recursion
RHS is a λ-abstraction

$$f = (\lambda F.\lambda n.\ (if\ (= n\ 0)\ 1\ (*\ n\ (F\ (-\ n\ 1)))))\ f$$

f is applied to itself

define

$$H = (\lambda F.\lambda n.\ (if\ (= n\ 0)\ 1\ (*\ n\ (F\ (-\ n\ 1)))))$$

then

$$f = H\ f$$

Y combinator

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

fixed point combinator
contains only bound variables.

ASIDE: Classification of functions

when called:
- return a value
- throw an exception
- exit (never return)
- ∞ loop (not return)