Lambda (λ) Calculus

What is it?

- very simple + powerful programming language
- only two concepts: function abstraction + application
- can be considered a universal machine code for programming languages (most naturally, functional ones)
  - we can describe constructs + techniques in other PLs in the λ calculus

Why do we need it?

- its simplicity makes it easier (than "real" PLs) to reason about
  - also easier to develop proof methods for it!
- if we can reduce some other language/program to the λ calculus, we can more easily prove things about that language/program

Grammar

also, parenthesized Es

$E ::= \text{var} \mid \text{function} \mid \text{application}$

- var is a variable name — we will stick to single letters

  $x, f, g, x_n, x'', \ldots$

- function (aka abstraction) $::= \lambda\, \text{var(s)} . E$ "body"

  $\lambda x.x$, $\lambda x.\lambda y.fxy$, $\lambda xy.\lambda f.\lambda g.f(gxy)$

- application $::= E\,E$

  $x\,y$, $f(gx)(hy)$, $(\lambda x.x)(\lambda x.x)$

Some more examples

$\lambda x.x$ — identity / id

$\lambda xy.x \equiv \lambda x.\lambda y.x$
$\lambda xy.y \equiv \lambda x.\lambda y.y$ — selection (fst, snd)

$\lambda fx.fx \equiv \lambda f.\lambda x.fx$ — apply

$\lambda x.xx$ — self-apply

$\lambda g.\lambda f.\lambda x.g(fx)$ — composition $(g \circ f)$

Associativity & Precedence

- function abstractions are right-associative; e.g.,

$$\lambda x.\lambda y.\lambda z.xyz \equiv \lambda x.(\lambda y.(\lambda z.xyz))$$

- function application is left-associative; e.g.,

$$fghx \equiv (((fg)h)x$$

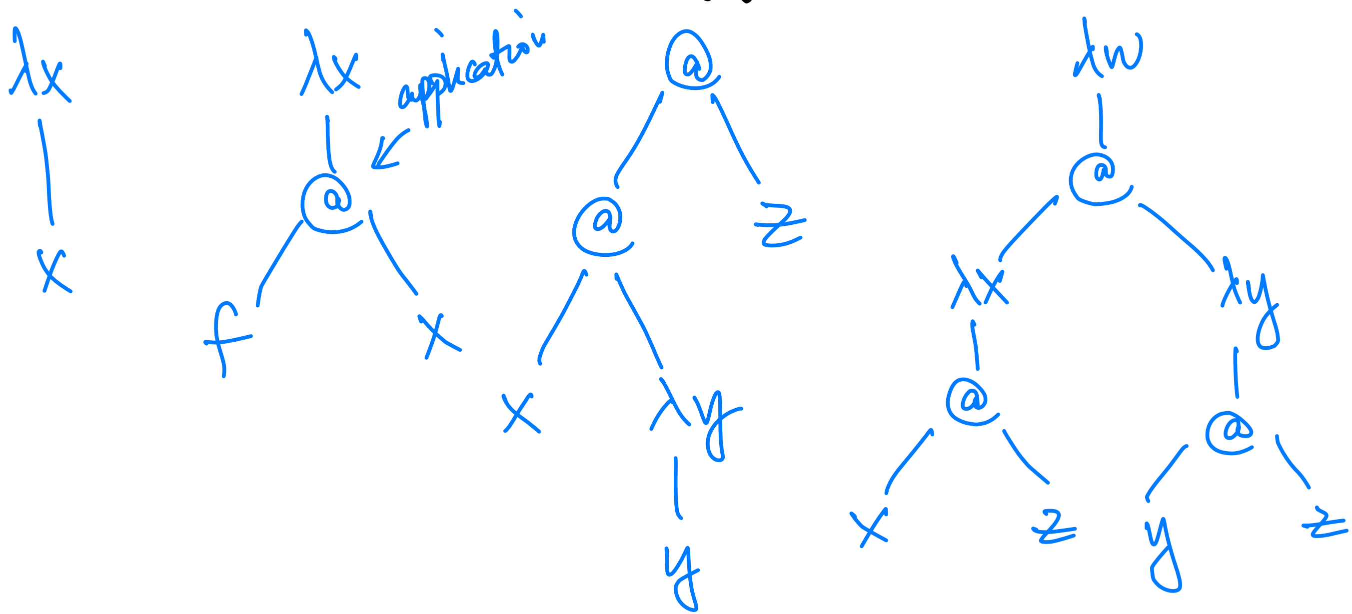
- function application has higher precedence than abstraction; e.g.,

$$\lambda x.xy\lambda z.xz \equiv \lambda x.((xy)(\lambda z.(xz))$$

λ calculus ASTs

– help us visualise relationships between sub-expressions

e.g. $\lambda x.x$ $\quad$ $\lambda x.fx$ $\quad$ $x(\lambda y.y)z$ $\quad$ $\lambda w.(\lambda x.xz)(\lambda y.yz)$



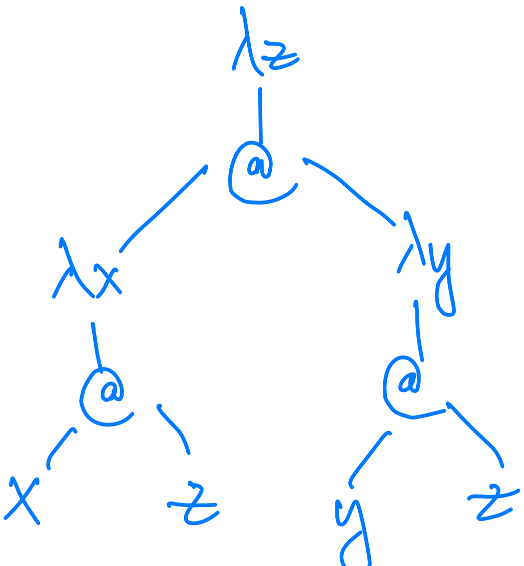
Free & Bound Variables

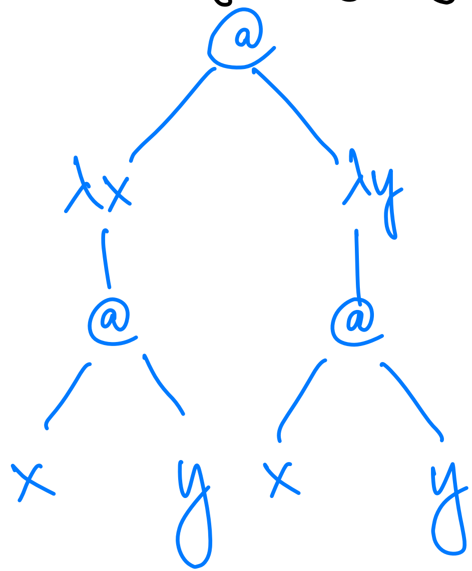
- function abstractions (λs) creates variable bindings
  - a variable in an expression is bound if it is preceded by a λ that binds it; otherwise, the variable is free

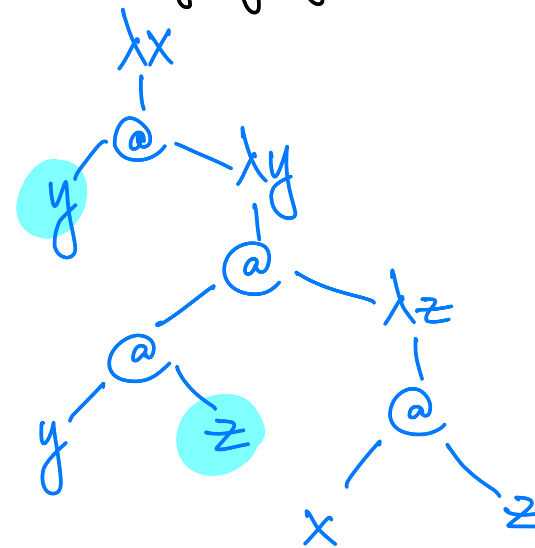
in a AST, if we can find a binding parent λ

– which variables are free and which are bound?

$\lambda z.(\lambda x.xz)(\lambda y.yz)$ $\quad$ $(\lambda x.xy)(\lambda y.xy)$ $\quad$ $\lambda x.y\lambda y.yz\lambda z.xz$







β-reduction (function application)

- we perform a β-reduction on an application of a λ-abstraction by substituting in the body of the λ the value of each argument for every corresponding instance of a variable bound by the λ

i.e., $(\lambda x.E)F \xrightarrow[\beta]{} [F/x]E$

substitute F for x in

examples :

$$(\lambda x.x)y \xrightarrow[\beta]{} y \qquad [y/x]x$$

$$(\lambda x.\lambda x.x)y \xrightarrow[\beta]{} \lambda x.x \qquad [y/x]\lambda x.x$$

new binding

different vars!

$$(\lambda x.xz)(\lambda y.y) \xrightarrow[\beta]{} (\lambda y.y)z \xrightarrow[\beta]{} z \qquad [\lambda y.y/x]xz$$

$$((\lambda x.\lambda y.yx)a)b \xrightarrow[\beta]{} \lambda y.ya \xrightarrow[\beta]{} ba$$

$$[a/x]\lambda y.yx \qquad [b/y]ya$$

$$(\lambda x.\lambda y.yx)y \xrightarrow[\beta]{} \cancel{\lambda y.yy} \qquad [y/x]\lambda y.yx$$

different meaning from the original function!

α-equivalence

- when two λ-abstractions use different bound variable names but otherwise have the same meaning, we say they are α-equivalent

- α-conversion is the process of renaming bound vars in λ-abstractions — it is semantic-preserving!

e.g., $\lambda x.x \xrightarrow[\alpha]{} \lambda y.y \xrightarrow[\alpha]{} \lambda z.z$

$$\lambda x.x \equiv \lambda y.y \equiv \lambda z.z$$

are these valid α-conversions? (are they α-equivalent?)

$$\lambda x.\lambda y.xy \overset{?}{\equiv} \checkmark \ \lambda a.\lambda b.ab$$

$$\lambda x.\lambda y.yyx \ \cancel{\overset{?}{\equiv}} \ \lambda g.\lambda f.ggf$$

$$\lambda x.x\lambda y.zy \ \cancel{\overset{?}{\equiv}} \ \lambda i.i\lambda j.kj$$

$$\lambda x.\lambda y.\lambda z.xzy \overset{?}{\equiv} \checkmark \ \lambda a.\lambda y.\lambda b.aby$$

$$\lambda x.\lambda y.yx \ \cancel{\overset{?}{\equiv}} \ \lambda y.\lambda y.yy$$

α-capture (variable capture)

- occurs when a β-reduction causes an a free variable to fall into the scope of a bound var w/ the same name

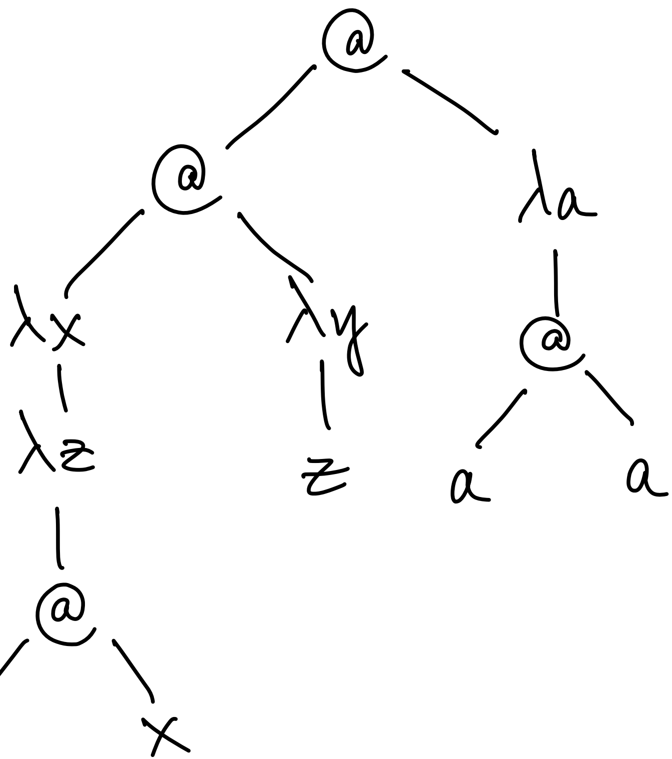
  e.g., $(\lambda x.\lambda y.yx)y \not\to \lambda y.yy$

- we prevent α-capture by α-converting the λ-abstraction so that there are no name clashes

  e.g. $(\lambda x.\lambda y.yx)y \xrightarrow[\alpha]{} (\lambda x.\lambda z.zx)y \xrightarrow[\beta]{} \lambda z.zy$

e.g. – convert this tree into an equivalent λ-expression

– simplify by performing as many β-reductions / α-conversions as possible



$$(\lambda x.\lambda z.zx)(\lambda y.z)(\lambda a.aa)$$

$$\equiv$$

$$(\lambda xz.zx)(\lambda y.z)(\lambda a.aa)$$

(thanks, Prof. Beekman!)

$$
\begin{tikzcd}
(\lambda x.\lambda z.zx)(\lambda y.z)(\lambda a.aa) \arrow[d, "\alpha"] & \\
(\lambda x.\lambda w.wx)(\lambda y.z)(\lambda a.aa) \arrow[d, "\beta\ {[\lambda y.z/x]\lambda w.wx}"] & z \\
(\lambda w.w\lambda y.z)(\lambda a.aa) \arrow[d, "\beta\ {[\lambda a.aa/w]w\lambda y.z}"] & \\
(\lambda a.aa)(\lambda y.z) \arrow[r, "\beta"', "{[\lambda y.z/a]aa}"] & (\lambda y.z)(\lambda y.z) \arrow[uu, "\beta\ {[\lambda y.z/y]z}"']
\end{tikzcd}
$$

"Eta" (Greek H)

η-reduction

- if x is not free in E, then $\lambda x. E x \xrightarrow[\eta]{} E$

  - think of this as anticipating a future β-reduction!

e.g. $(\lambda x. g x) y \xrightarrow[\beta: [y/x] g x]{} g y$

$\xrightarrow[\eta]{} (g) y$

e.g. $(\lambda y . \lambda x . y x) w z \xrightarrow[\beta]{} (\lambda x . w x) z \xrightarrow[\beta]{} w z$

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

e.g. $\lambda y . \lambda x . y x \xrightarrow[\eta]{} \lambda y . y$

– demonstrates equivalence w/o application / β-reduction

Normal Form

- a reducible expression (aka redex) is any expression to which we can immediately apply a β-reduction
- an expression w/ no redexes is in normal form
  - not all expressions can be reduced to normal form!

    e.g., $(\lambda x.xx)(\lambda x.xx) \xrightarrow[\beta]{} (\lambda x.xx)(\lambda x.xx)$
  - do we always need to reduce to normal form?
    - "real" programming languages typically don't! (why?)

Weak Head Normal Form (WHNF)

- If the root node of the AST is a λ-abstraction, we do not need to reduce any further — this is WHNF

  e.g., $\lambda x.(\lambda y.y)(\lambda z.z)$ is in WHNF

  for normal form: $\xrightarrow{\beta} \lambda x.\lambda z.z$

- intuition: wait until function is applied to evaluate its body
- note: we will often prefer to evaluate to normal form to better understand the meaning of a λ-expression

Evaluation Order

- If we have multiple redexes, in what order do we perform β-reductions?

* **Applicative order evaluation**: always reduce the leftmost, innermost redex first
  i.e., evaluate args before applying function

* **Normal order evaluation**: always reduce the leftmost, outermost redex first
  i.e., apply function before evaluating args

Evaluation Order

* **Applicative order evaluation**:

$$\frac{E_2 \xrightarrow[\beta]{} E_2'}{(\lambda x. E_1) E_2 \xrightarrow[\beta]{} (\lambda x. E_1) E_2'}$$

call-by-value / eager evaluation

* **Normal order evaluation**:

$$\frac{}{(\lambda x. E_1) E_2 \xrightarrow[\beta]{} [E_2/x] E_1}$$

call-by-name / lazy evaluation

e.g. $(\lambda x.\lambda f.fx)(\lambda z.z)((\lambda q.q)(\lambda r.r))$

applicative order:

$(\lambda f.f\lambda z.z)((\lambda q.q)(\lambda r.r))$

$(\lambda f.f\lambda z.z)\lambda r.r$

$\lambda r.r\lambda z.z$

$\lambda z.z$

normal:

$(\lambda f.f\lambda z.z)((\lambda q.q)(\lambda r.r))$

$((\lambda q.q)(\lambda r.r))(\lambda z.z)$

$\lambda r.r\lambda z.z$

$\lambda z.z$

If a λ-expression has a normal form,

- performing β-reductions in normal order will get us there
- if applicative order evaluation terminates, it will yield the same result as normal order

may not!

e.g. $(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$

applicative:

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

$$\downarrow \beta$$

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

$$\downarrow$$

$$\vdots$$

normal:

$$\xrightarrow{\beta} y$$

But!

- applicative order often has fewer reductions

  e.g., $(\lambda x.\underbrace{xxx\ldots}_{\text{just use result}})(\underbrace{(\ )(\ )(\ )\ldots}_{\text{evaluate once}})$

- normal order may be more efficient

  e.g., $(\lambda x.y)(\underbrace{(\ )(\ )(\ )\ldots}_{\text{never evaluated!}})$

Church-Rosser Theorem

- the order in which reductions are applied to λ expressions does not change the eventual result
    - guarantees that normal form is unique!
- more formally, that β-reduction satisfies the diamond property:
    - if $E_1 \xrightarrow[\beta]{}^{*} E_2$ and $E_1 \xrightarrow[\beta]{}^{*} E_3$, there exists some E_4 such that $E_2 \xrightarrow[\beta]{}^{*} E_4$ and $E_3 \xrightarrow[\beta]{}^{*} E_4$

$$\begin{tikzcd} & E_1 \arrow[dl, "\beta^*"'] \arrow[dr, "\beta^*"] & \\ E_2 \arrow[dr, "\beta^*"'] & & E_3 \arrow[dl, "\beta^*"] \\ & E_4 & \end{tikzcd}$$

(proof is not trivial)

Outline of Proof

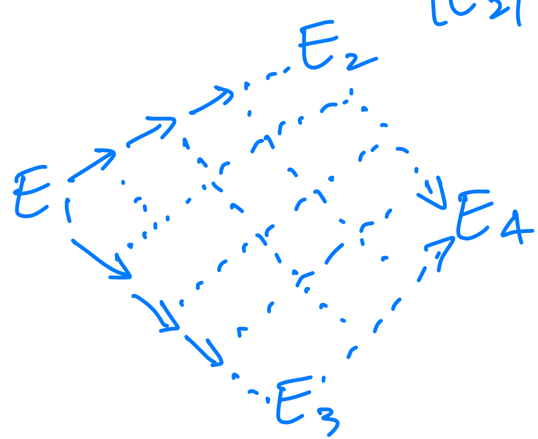
single step

Lemma: if $E_1 \xrightarrow[\beta]{} E_2$ and $E_1 \xrightarrow[\beta]{} E_3$, there exist some E_4 s.t. $E_2 \xrightarrow[\beta]{*} E_4$ and $E_3 \xrightarrow[\beta]{*} E_4$

proof of lemma:

$$
\begin{tikzcd}
 & (\lambda x.E_1)E_2' \arrow[dr, "*"] & \\
(\lambda x.E_1)E_2 \arrow[ur] \arrow[dr] & & {[E_2'/x]E_1} \\
 & {[E_2/x]E_1} \arrow[ur, "*"'] &
\end{tikzcd}
$$

outline of proof (not trivial):



(inductive proof)

Corollary: if $E_1 \xrightarrow[\beta]{*} E_2$ and $E_1 \xrightarrow[\beta]{*} E_3$, where both E_2 and E_3 are in normal form, then $E_2 = E_3$

i.e., when a λ-expression has a normal form, it is unique.

Data representation

- the "pure" λ-calculus only has functions as values
  - sometimes we cheat and extend the syntax to include common values (e.g., integers) and operators

    e.g., $((\lambda x . \lambda y . x + y)\ 10)\ 20 \Downarrow 30$

  - we can also encode values such as integers and Booleans (and related operators) by mapping them to functions!

Boolean Operators + Values

- we want to define functions that represent:

TRUE, FALSE, IF, NOT, AND, OR

where:

IF TRUE E_1 E_2 = E_1

IF FALSE E_1 E_2 = E_2

NOT TRUE = FALSE

NOT FALSE = TRUE

AND TRUE TRUE = TRUE

" FALSE TRUE = FALSE

...

OR TRUE FALSE = TRUE

" FALSE FALSE = FALSE

...

clue to implementation:

TRUE picks the first of two,

FALSE picks the second

Boolean Operators + Values

$$\text{TRUE} \equiv \lambda x.\lambda y.x$$

$$\text{FALSE} \equiv \lambda x.\lambda y.y$$

$$\text{IF} \equiv \lambda b.\lambda e_1.\lambda e_2.\, b\, e_1\, e_2$$

$$\text{NOT} \equiv \lambda b.b\,(\lambda x.\lambda y.y)\,(\lambda x.\lambda y.x) \equiv \lambda b.b\ \text{FALSE}\ \text{TRUE}$$

$$\text{AND} \equiv \lambda b_1.\lambda b_2.\, b_1\, b_2\ \text{FALSE}$$

$$\text{OR} \equiv \lambda b_1.\lambda b_2.\, b_1\ \text{TRUE}\ b_2$$

Church Numerals

- one way to encode integers in the λ-calculus
- idea: integer n represents n repeated applications of a function f to an argument x

$$0 \equiv \lambda f.\lambda x.x$$

$$1 \equiv \lambda f.\lambda x.fx$$

$$2 \equiv \lambda f.\lambda x.f(fx)$$

$$\vdots$$

Church Numerals

- how do we increment (+1) a Church numeral?
  - define INC, where INC 0 = 1 , INC 1 = 2, ...

$$\text{INC} \equiv \lambda n.\lambda f.\lambda x.\underbrace{f(f(f(\ldots(f}_{n+1 \text{ times}}\, x))))$$

$$\equiv \lambda n.\lambda f.\lambda x.f(n\,f\,x)$$

(try DEC for a challenge!)

Church Numerals

- how do we add Church Numerals?
  - define ADD, where ADD M N = the $(M+N)^{th}$ Church Numeral

$$\text{ADD} \equiv \lambda m.\lambda n.\lambda f.\lambda x.\, m f\,(n f x)$$

(try MULT for a challenge)

or

$$\text{ADD} \equiv \lambda m.\lambda n.\, \underbrace{m\ \text{INC}}_{\text{apply INC } m \text{ times to } n}\ n$$

Recursion

- how might we implement the following Racket function in the λ-calculus?

```
(define (f x) (f (+ x 1)))
```

what is f?

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

but we must pass in itself, so...

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

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

but now f needs a copy of itself

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

$$(\lambda f.\lambda x.ff(\mathrm{INC}\ x))(\lambda f.\lambda x.ff(\mathrm{INC}\ x))\ 0$$

$$(\lambda x.(\lambda f.\lambda x.ff(\mathrm{INC}\ x))(\lambda f.\lambda x.ff(\mathrm{INC}\ x))(\mathrm{INC}\ x))\ 0$$

$$(\lambda f.\lambda x.ff(\mathrm{INC}\ x))(\lambda f.\lambda x.ff(\mathrm{INC}\ x))\cancel{(\mathrm{INC}\ 0)} \rightarrow 1$$

$$(\lambda x.(\lambda f.\lambda x.ff(\mathrm{INC}\ x))(\lambda f.\lambda x.ff(\mathrm{INC}\ x))(\mathrm{INC}\ x))\ 1$$

$$(\lambda f.\lambda x.ff(\mathrm{INC}\ x))(\lambda f.\lambda x.ff(\mathrm{INC}\ x))\cancel{(\mathrm{INC}\ 1)} \rightarrow 2$$

$$\vdots$$

The Y-Combinator

- def. combinator: a HOF that uses only function application to compute a result

- we can extract this pattern of repeated function applications:

  $f(f(f(f \ldots$

- define Y, where $Yf = f(Yf) = f(f(Yf)) = \ldots$

$$Y \equiv \lambda f.(\lambda x. f(xx))(\lambda x. f(xx))$$

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

try $Y\,F$

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

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

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

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

$$\vdots$$

Termination?

- is it possible for $(Y\,F)$ to terminate?

$$Y\,F = F(Y\,F) = F(F(Y\,F)) \ldots$$

- Yes! if F does not evaluate its argument.

  e.g. $F \equiv \lambda x.\,T$

  - but only under call-by-name semantics!

    - what happens under call-by-value?

Using Y

e.g. use Y to implement the recursive factorial function:

$$FACT \approx \lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n \times FACT(n-1)$$

$$FACT' \equiv \lambda f.\lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n \times f(n-1)$$

$$FACT \equiv Y\, FACT'$$

$$= (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\, FACT'$$

$$= \lambda x.FACT'(xx)(\lambda x.FACT'(xx))$$

$$= FACT'(\lambda x.FACT'(xx))(\lambda x.FACT'(xx))$$

$$\mathrm{FACT} \approx \lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n \times \mathrm{FACT}(n-1)$$

$$\mathrm{FACT}' \equiv \lambda f.\lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n \times f(n-1)$$

$$\begin{aligned}
\mathrm{FACT} &\equiv Y\ \mathrm{FACT}' \\
&= \lambda x.\mathrm{FACT}'(x x)(\lambda x.\mathrm{FACT}'(x x)) \\
&= \mathrm{FACT}'\underbrace{(\lambda x.\mathrm{FACT}'(x x))(\lambda x.\mathrm{FACT}'(x x))} \\
&= \mathrm{FACT}'(Y\ \mathrm{FACT}') \\
&= \mathrm{FACT}'\ \mathrm{FACT}
\end{aligned}$$

$$FACT' \equiv \lambda f.\lambda n.\text{if } n=0 \text{ then } 1 \text{ else } n \times f(n-1)$$

$$FACT' \; FACT$$

$$= \lambda n.\text{if } n=0 \text{ then } 1 \text{ else } n \times FACT(n-1)$$

- we have seen that we can represent different types, control constructs, and recursion in the λ-calculus

- some semantic patterns are a bit harder to translate, however...
  - exceptions
  - non-deterministic algorithms
  - generators & coroutines

Continuations

- a concept / technique that we can use to explicitly express the control-flow of expressions / programs
  - very handy for building control-or   semantics!
- at any point in the evaluation of a program, a continuation represents the rest of the program

e.g., in $\lambda x.\, h(g(f\, x))$ what is the continuation after evaluating the subexpression $(f\, x)$?

"semantic brackets"

$$h(g\, [\![\,]\!])$$

- how could we represent this continuation ?

$$k = \lambda v.\, h(g\, v)$$

- how do we "use" this continuation ?

$$k\,(f\, x)$$