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 \( \lambda \) calculus, we can more easily prove things about that language/program.
Grammar

\[ E ::= \text{var} \mid \text{function} \mid \text{application} \]

- \text{var} is a variable name — we will stick to single letters \( x, f, g, x_n, x'' \).

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

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

- \text{application} ::= EE

\( x\,y, f(gx)(hy), (\lambda x.x)(\lambda x.x) \)
Some more examples

\( \lambda x. x \)
\( \lambda xy. x \equiv \lambda x. ly. x \)
\( \lambda xy. y \equiv \lambda x. ly. y \)
\( \lambda f x. fx \equiv \lambda f. lx. fx \)
\( \lambda x. xx \)
\( \lambda g. \lambda f. lx. g(fx) \)

identity/id

\( \{ \text{selection} (\text{fst, snd}) \) apply

self-apply

composition (gof)
Associativity & Precedence

- Function abstractions are right-associative; e.g.,
  \[ \lambda x. \lambda y. \lambda z. xyz = \lambda x. (\lambda y. (\lambda z. xyz)) \]

- Function application is left-associative; e.g.,
  \[ fghx = (((fg)h)x) \]

- Function application has higher precedence than abstraction; e.g.,
  \[ \lambda x. x y \lambda z. x z = \lambda x. (((x y)(\lambda z. (x z))) \]
\[ \text{\textit{\lambda} calculus ASTs} \]

- Help us visualise relationships between sub-expressions

\[ \text{e.g. } \lambda x.x \quad \lambda x.fx \quad x(\lambda y.y)z \quad \lambda w.(\lambda x.xz)(\lambda y.yz) \]
Free & Bound Variables

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

in a AST, if we can find a binding parent \(\lambda\)
which variables are free and which are bound?
B-reduction (function application)

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

i.e.,

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

substitute $F$ for $x$ in
examples:\n\[
\begin{align*}
(\lambda x. x)y & \xrightarrow{\beta} y \\
(\lambda x. \lambda x. x)y & \xrightarrow{\beta} \lambda x. x \\
(\lambda x. xz)(\lambda y. y) & \xrightarrow{\beta} (\lambda y. y)z \\
(\lambda x. \lambda y. yx)ab & \xrightarrow{\beta} \lambda y. ya \\
(\lambda x. \lambda y. yx)y & \xrightarrow{\beta} \lambda y. yx
\end{align*}
\]
\[\text{different vars!}\]
\[\text{new binding}\]
\[\text{different meaning from the original function!}\]
**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!

\[ \forall x, y, \ z. \quad \lambda x. x \rightarrow_{\alpha} \lambda y. y \rightarrow_{\alpha} \lambda z. z \]

\[ \lambda x. x \equiv \lambda y. y \equiv \lambda z. z \]
are these valid α-conversions? (are they α-equivalent?)

\[ \lambda x. \lambda y. x y \equiv \lambda a. \lambda b. a b \]

\[ \lambda x. \lambda y. y x \equiv \lambda g. \lambda f. g f \]

\[ \lambda x. \lambda y. z y \equiv \lambda i. \lambda i j. k j \]

\[ \lambda x. \lambda y. x z. x z y \equiv \lambda a. \lambda y. \lambda b. a b \]

\[ \lambda x. \lambda y. y x \not\equiv \lambda y. \lambda y. y y \]
\( \alpha \)-capture (variable capture)

- occurs when a \( \beta \)-reduction causes an a-free variable to fall into the scope of a bound var \( y \) of the same name

  \[ \text{e.g., } (\lambda x. \lambda y. yx) y \rightarrow \lambda y. yy \]

- we prevent \( \alpha \)-capture by \( \alpha \)-converting the \( \lambda \)-abstraction so that there are no name clashes

  \[ \text{e.g. } (\lambda x. \lambda y. yx) y \xrightarrow{\alpha} (\lambda x. \lambda z. z x) y \xrightarrow{\beta} \lambda z. z y \]
e.g. - convert this tree into an equivalent λ-expression
- simplify by performing as many β-reductions/α-conversions as possible

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

(Thanks, Prof. Beekman!)
\[
\frac{\lambda x.\lambda z.\, z\, x}{\lambda \alpha} \frac{\lambda x.\lambda w.\, w\, x}{\beta \left[ \lambda y.\, z / x \right] \lambda w.\, w x} \frac{\lambda w.\, w \, \lambda y.\, z}{\beta \left[ \lambda a.\, a a / w \right] \lambda w \, \lambda y.\, z} \frac{\lambda a.\, a a}{\beta \left[ \lambda y.\, z / a \right] a} \frac{\lambda y.\, z}{\left( \lambda y.\, z \right) \left( \lambda y.\, z \right)} \]

"Eta" (Greek Η)

\[ \eta \text{-reduction} \]

- If \( x \) is not free in \( E \), then \[ \lambda x. E x \xrightarrow{\eta} E \]

- Think of this as anticipating a future \( \beta \)-reduction!

E.g. \( (\lambda x.g x)y \xrightarrow{\beta} [y/x]g y \)

\[ \eta \xrightarrow{} (g)y \]
e.g. \((\lambda y. \lambda x. yx)wz \xrightarrow{\beta} (\lambda x. wx)z \xrightarrow{\beta} wz\)

\(\eta \rightarrow (\lambda y. y)wz \xrightarrow{\beta} wz\)

e.g. \(\lambda y. \lambda x. yx \xrightarrow{\eta} \lambda y. y\)

- demonstrate equivalence w.r.t. application/\(\beta\)-reduction
Normal Form

- a reducible expression (aka redex) is any expression to which we can immediately apply a $\beta$-reduction

- an expression up no redexes is in normal form

- not all expressions can be reduced to normal form!

  e.g., $(\lambda x . xx)(\lambda x . xx) \overset{\beta}{\rightarrow} (\lambda x . xx)(\lambda x . xx)$

- 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:  \[ \beta \rightarrow \lambda x. \lambda z. z \]

- 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:

\[ E_2 \xrightarrow{p} E'_2 \]

\[ (\lambda x. E_1) E_2 \xrightarrow{p} (\lambda x. E_1) E'_2 \]

call-by-value / eager evaluation

*Normal order evaluation:

\[ (\lambda x. E_1) E_2 \xrightarrow{p} [E_2/x] E_1 \]

call-by-name / lazy evaluation
e.g. \((\lambda x.\lambda f.f x)(\lambda z.z)((\lambda g.g)(\lambda r.r))\)

**Applicative order:**

\([\lambda f.\lambda z.z]((\lambda g.g)(\lambda r.r))\)

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

\(\lambda r. r \lambda z.z\)

\(\lambda z.z\)

**Normal:**

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

\(((\lambda g.g)(\lambda r.r))(\lambda z.z)\)

\(\lambda r. r \lambda z.z\)

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

application: \(\beta\)

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

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

\(\vdots\)

\(\vdots\)
But!

- applicative order often has fewer reductions
  e.g., \((\lambda x. xxx \ldots)(()())()()\ldots\)
    
  \[\text{just use result} \quad \text{evaluate once}\]

- normal order may be more efficient
  e.g., \((\lambda x. y)(()())()()\ldots\)

  \[\text{never evaluated!}\]
Church-Rosser Theorem

- the order in which reductions are applied to $\lambda$ expressions does not change the eventual result.

- guarantees that normal form is unique!

- more formally, that $\beta$-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 $E_3 \xrightarrow{\beta}^* E_4$

(proof is non-trivial)
**Outline of Proof**

Lemma: if $E_1 \xRightarrow{\beta} E_2$ and $E_1 \xRightarrow{\beta} E_3$, there exists some $E_4$ s.t. $E_2 \xRightarrow{\beta^*} E_4$ and $E_3 \xRightarrow{\beta^*} E_4$

Proof of Lemma: $(\lambda x. E_1)E_2 \xrightarrow{\beta} [E_2/x]E_1 \xrightarrow{\beta^*} [E_2'/x]E_1$

Outline of Proof (not trivial): $E_1 \xrightarrow{\beta} E_4$ (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
  \" FALSE FALSE = FALSE
OR TRUE FALSE = TRUE
  \" FALSE FALSE = FALSE
\ldots

\textit{due to implementation:}

TRUE picks the first of two,
FALSE picks the second
Boolean Operators + Values

**TRUE** = \( \lambda x. \lambda y. x \)

**FALSE** = \( \lambda x. \lambda y. y \)

**IF** = \( \lambda b. \lambda e_1. \lambda e_2. b \ e_1 \ e_2 \)

**NOT** = \( \lambda b. b \ (\lambda x. \lambda y. y) \ (\lambda x. \lambda y. x) \equiv \lambda b. b \ \text{FALSE TRUE} \)

**AND** = \( \lambda b_1. \lambda b_2. b_1 \ b_2 \ \text{FALSE} \)

**OR** = \( \lambda b_1. \lambda b_2. b_1. \text{TRUE} \ b_2 \)
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

\[
\begin{align*}
0 & \equiv \lambda f. \lambda x. x \\
1 & \equiv \lambda f. \lambda x. fx \\
2 & \equiv \lambda f. \lambda x. f(fx) \\
& \vdots
\end{align*}
\]
Church Numerals

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

\[ INC \equiv \lambda n. \lambda f. \lambda x. f(f(f(\ldots(fx)))) \]
\[ \text{for } n+1 \text{ times} \]

\[ \equiv \lambda n. \lambda f. \lambda x. f(nfx) \quad \text{(try DEC for a challenge!)} \]
Church Numerals

- how do we add Church Numerals?

- define \( \text{ADD} \), where \( \text{ADD} \ M \ N = \text{the } (M+N)_{\text{th}} \text{ Church Numeral} \)

\[
\text{ADD} \equiv \lambda m. \lambda n. \lambda f. \lambda x. m f (n f x)
\]

or

\[
\text{ADD} \equiv \lambda m. \lambda n. \, m \, \text{INC} \, n
\]

apply INC \( m \) times to \( n \)

(try \text{MULT} for a challenge)
Recursion

- how might we implement the following Racket function in the \( \lambda \)-calculus?

\[
(\text{define } (f \ x) (f (+ \ x \ 1)))
\]

what is \( f \)?

\[
\lambda x. f \ (\text{inc} \ x)
\]

but we must pass in itself, so...

\[
(\lambda f. \lambda x. f \ (\text{inc} \ x))(\lambda f. \lambda x. f \ (\text{inc} \ x))
\]

(\( \lambda f. \lambda x. ff \ (\text{inc} \ x) \))(\( \lambda f. \lambda x. ff \ (\text{inc} \ x) \))
\[
(\lambda f. \lambda x. ff(inc x))(\lambda f. \lambda x. ff(inc x))\circ (\lambda x. (\lambda f. \lambda x. ff(inc x))(\lambda f. \lambda x. ff(inc x))(inc x))\circ (\lambda f. \lambda x. ff(inc x))(\lambda f. \lambda x. ff(inc x))(inc x)\circ 1
\]
\[
(\lambda x. (\lambda f. \lambda x. ff(inc x))(\lambda f. \lambda x. ff(inc x))(inc x))\circ 1
\]
\[
(\lambda f. \lambda x. ff(inc x))(\lambda f. \lambda x. ff(inc x))(inc x)\circ 2
\]
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:
  \[ f \circ f \circ f \circ f \ldots \]

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

\[ Y = \lambda f. (\lambda x. f(xx))(\lambda x. f(xx)) \]
\[
Y = \lambda f. (\lambda x. f(x x))(\lambda x. f(x x))
\]

\[=
(\lambda x. F(x x))(\lambda x. F(x x))
\]

\[=
F((\lambda x. F(x x))(\lambda x. F(x x)))
\]

\[=
F(F((\lambda x. F(x x))(\lambda x. F(x x))))
\]

\[=
F(F(F((\lambda x. F(x x))(\lambda x. F(x x)))))
\]

\[= \ldots
\]
Termination?

- is it possible for \((YF)\) to terminate?

\[ YF = F(YF) = F(F(YF)) \ldots \]

- Yes! if \(F\) does not evaluate its argument.

- e.g. \(F = \lambda x. T\)

- 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:

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

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

$$\text{FACT} = Y \text{FACT}'$$

\[ = (\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))) \text{FACT}' \]

\[ = \lambda x. \text{FACT}'(xx) (\lambda x. \text{FACT}'(xx)) \]

\[ = \text{FACT}'(\lambda x. \text{FACT}'(xx)) (\lambda x. \text{FACT}'(xx)) \]
FACT ≡ \lambda n. \text{if } n=0 \text{ then } 1 \text{ else } n \times FACT(n-1)

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

FACT ≡ Y FACT'

= \lambda x. FACT'(x \times)(\lambda x. FACT'(x \times))

= FACT'(\lambda x. FACT'(x \times))(\lambda x. FACT'(x \times))

= FACT'(Y FACT')

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

\[
FACT' \text{ FACT } = \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{FACT}(n-1)
\]
- 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 \((fx)\)?

- how could we represent this continuation?

\[
\begin{align*}
  h (g [\, [\, ]]) \\
  k = \lambda v. h (g v)
\end{align*}
\]

- how do we "use" this continuation?

\( k (fx) \)