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

  \( \lambda x.x \), \( \lambda x.\lambda y.fx.y \), \( \lambda y.\lambda f.\lambda g.f(gxy) \)

- \text{application} ::= EE

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

\( \lambda x. x \)
\( \lambda xy. x = \lambda x. \lambda y. x \)
\( \lambda xy. y = \lambda x. \lambda y. y \)
\( \lambda fx. fx = \lambda f. \lambda x. fx \)
\( \lambda x. xx \)
\( \lambda g. \lambda f. \lambda x. g(fx) \)

identity/id

\{ selection \ (fst, snd) \}

apply

self-apply

composition \ (gof) \)
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.,
  \[ \lambda x. x y \lambda z. x z \equiv \lambda x. ((x y)(\lambda z. (x z))) \]
\[ \lambda \text{calculus ASTs} \]

- 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.
- which variables are free and which are bound?

\( \lambda z. (\lambda x. x z)(\lambda y. y z) \)  
\( (\lambda x. x y)(\lambda y. x y) \)  
\( \lambda x. y k y. y z = \lambda z. x z \)
**$\beta$-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 $\lambda$

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

substitute $F$ for $x$ in
Examples:

\[(\lambda x.x)y \xrightarrow{\beta} y\]
\[(\lambda x.\lambda x.x)y \xrightarrow{\beta} \lambda x.x \quad [y/x] \lambda x.x \quad \text{new binding} \]
\[(\lambda x.xz)(\lambda y.y) \xrightarrow{\beta} (\lambda y.y)z \xrightarrow{\beta} z\]
\[(\lambda x.\lambda y.yx)ab \xrightarrow{\beta} \lambda y.ya \xrightarrow{\beta} ba\]
\[(\lambda x.\lambda y.yx)y \xrightarrow{\beta} \lambda y.yy \quad [y/x] \lambda y.yx \quad \text{different meaning from the original function!}\]
\( \alpha \)-equivalence

- When two \( \lambda \)-abstractions use different bound variable names but otherwise have the same meaning, we say they are \( \alpha \)-equivalent.

\( \alpha \)-conversion is the process of renaming bound vars in \( \lambda \)-abstractions — it is semantic-preserving!

\[ \lambda x.x \xrightarrow{\alpha} \lambda y.y \xrightarrow{\alpha} \lambda z.z \]

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

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

\[ \lambda x. \lambda y. y y x \not\equiv \lambda g. \lambda f. q g f \]

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

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

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

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

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

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

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

\[
(λx.λz.zx)(λy.z)(λa.aa)
\]

(thanks, Prof. Beekman!)
\[(\lambda x.\lambda z. zx) (\lambda y. z) (\lambda a. aa)\]
\[\downarrow \beta \quad [\lambda y. z/x] \lambda w. wx\]
\[(\lambda w. w y. z) (\lambda a. aa)\]
\[\downarrow \beta \quad [\lambda a. aa/w] \lambda w y. z\]
\[\lambda a. aa)(\lambda y. z)\]
\[\downarrow \beta \quad [\lambda y. z/a] \lambda a. aa\]
\[(\lambda y. z)(\lambda y. z)\]
Normal Form

-a reducible expression (aka redex) is any expression to which we can immediately apply a $\beta$-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., \( \lambda x \cdot (\lambda y \cdot y) (\lambda z \cdot z) \) is in WHNF

  for normal form: \( \beta \rightarrow \lambda x \cdot \lambda z \cdot z \)

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

Appendive order:

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

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

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

\(\lambda z. z\)

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

applicative: \(\beta\)  
\[(\lambda x. y)(\lambda x. xx)(\lambda x. xx)\]  
\(\beta\)  
\[(\lambda x. y)(\lambda x. xx)(\lambda x. xx)\]  

normal: \(\beta\)  
\(y\)
But!

- Applicative order often has fewer reductions.

  e.g., \((\lambda x. \text{xxx} \ldots) () () () () \ldots\)

  \[\text{just use result \hspace{1cm} evaluate once}\]

- 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 λ expressions does not change the eventual result

- guarantees that normal form is unique!

- more formally, that β-reduction satisfies the diamond property:

  if $E_1 \beta \rightarrow^* E_2$ and $E_1 \beta \rightarrow^* E_3$, there exists some $E_4$ such that $E_2 \beta \rightarrow^* E_4$ and $E_3 \beta \rightarrow^* E_4$

(proof is not trivial)
Lemma: if $E_1 \Rightarrow E_2$ and $E_1 \Rightarrow E_3$, then exist some $E_4$ s.t. $E_2 \Rightarrow E_4$ and $E_3 \Rightarrow E_4$

Proof of lemma: $(\lambda x.E_1)E_2 \Rightarrow (\lambda x.E_1)E_2'$ 

Outline of proof: $E 

\Rightarrow E_1 \Rightarrow 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" $\lambda$-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:

\[
\begin{align*}
  \text{IF } \text{TRUE} & \quad E_1, E_2 = E_1 \\
  \text{IF } \text{FALSE} & \quad E_1, E_2 = E_2 \\
  \text{NOT } \text{TRUE} & \quad = \text{FALSE} \\
  \text{NOT } \text{FALSE} & \quad = \text{TRUE} \\
  \text{AND } \text{TRUE} \text{ TRUE} & \quad = \text{TRUE} \\
  \text{AND } \text{FALSE} \text{ TRUE} & \quad = \text{FALSE} \\
  \text{OR } \text{TRUE} \text{ FALSE} & \quad = \text{TRUE} \\
  \text{OR } \text{FALSE} \text{ FALSE} & \quad = \text{FALSE}
\end{align*}
\]

... due to implementation:

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

\[ \text{TRUE} = \lambda x. \lambda y. x \]
\[ \text{FALSE} = \lambda x. \lambda y. y \]
\[ \text{IF} = \lambda b. \lambda e_1. \lambda e_2. b \ e_1 \ e_2 \]
\[ \text{NOT} = \lambda b. \ b (\lambda x. \lambda y. x) (\lambda x. \lambda y. y) = \lambda b. b \ \text{FALSE} \ \text{TRUE} \]
\[ \text{AND} = \lambda b_1. \lambda b_2. \ b_1 \ b_2 \ \text{FALSE} \]
\[ \text{OR} = \lambda b_1. \lambda b_2. \ b_1. \ \text{TRUE} \ b_2 \]


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

\[
0 \equiv \lambda f.\lambda x. x \\
1 \equiv \lambda f.\lambda x. fx \\
2 \equiv \lambda f.\lambda x. f(fx) \\
\vdots
\]


Church Numerals

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

\[ INT \equiv \lambda n. \lambda f. \lambda x. f(f(f(...(f x)))) \]

\( n+1 \) times

\[ \equiv \lambda n. \lambda f. \lambda x. f(n f x) \]

(try DEC for a challenge!)
Church Numerals

- how do we add Church Numerals?

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

$$ADD \equiv \lambda m. \lambda n. \lambda f. \lambda x. m \ f \ (n \ f \ x)$$

or

$$ADD \equiv \lambda m. \lambda n. m \ INC \ n$$

apply $INC$ $m$ times to $n$ (try $MULT$ 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...

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

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

but now \(f\) needs a copy of itself

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

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

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

\[\lambda f. \lambda x. f\ (\text{inc} x)\]
\[(\lambda f. \lambda x. f f (\text{inc } x))(\lambda f. \lambda x. f f (\text{inc } x)) \circ \ldots \circ (\lambda f. \lambda x. f f (\text{inc } x))(\lambda f. \lambda x. f f (\text{inc } x)) (\text{inc } 0)\]
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 \( \textbf{Y} \), where \( \textbf{Y} f = f(\textbf{Y} f) = f(f(\textbf{Y} f)) = \ldots \)

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

\[ \text{try } YF \]

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

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

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

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

\[ \vdots \]
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. $F \equiv \lambda x. T$

  - but only under call-by-name semantics!
  - what happens under call-by-value?
- Infinitely more combinators exist!

- We can define arbitrary types using pure $\lambda$-calculus

- We can even represent $\lambda$-calculus in $\lambda$-calculus

- $\lambda$-calculus is the basis for many "real" programming languages (e.g., Haskell, Lisp, Racket, ML), and can serve as a "universal functional machine code"