Untyped Lambda Calculus
Original $\lambda$-CALCULUS SYNTAX

$e$ is a lambda expression, or lambda term.

$$e ::= x \quad \text{(a variable)}$$
$$\mid \ \lambda x.e \quad \text{(a nameless function)}$$
$$\mid e \ e \quad \text{(function application)}$$

$$v ::= \lambda x.e \quad \text{(only functions can be values)}$$

Above is a BNF (Backus Naur Form) that specifies the abstract syntax of the language

[“\” will be written “λ” in a nice font]

Note the above is inductive definition: $e$, $x$ are meta-variables

Expressions which are functions are also called lambda abstractions.
Essentially every full-scale programming language has some notion of **function**

- the (pure) lambda calculus is a language composed **entirely** of functions
- we use the lambda calculus to study the essence of computation
- it is just as fundamental as Turing Machines
More syntax

- the identity function:
  - \( \lambda x.x \)
    - Mathematically equivalent to: \( f(x) = x \).

- 2 notational conventions:
  - applications associate to the left (like in):
    - “\( y z x \)” is “\( (y z) x \)”
  - the body of a lambda extends as far as possible to the right:
    - “\( \lambda x. \lambda x z x \)” is “\( \lambda x. (\lambda z (x z x)) \)”
Names and Denotable Objects

- Name is a sequence of characters used to represent or denote an object.
- “Object” is used in the general sense. The most common object we see in this course is a variable.
- E.g.,
  \`\`foo.\`foo \`\`bar.\`foo bar foo
**Names and Denotable Objects**

- A name and the object it denote are NOT the same thing!
- A name is merely a “character string”.
- An object can have multiple names – “aliasing”.
- A name can denote different objects at different times.
- “variable bar” means “the variable with the name bar”.
- “function foo” means “the function with the name foo”.

**Binding**

- *Binding* is an association between a name and the denotable object it represents
  - *Static binding*: during language design, compile time
  - *Dynamic binding*: during run time

- The *scope* of a name is the region of a program which can access the name binding.

- The *lifetime* of a name refers to the time interval during which the name remains *bound*. 
Scopes in λ-calculus

- $\langle x. e \rangle$: the scope of $x$ is the term $e$ ($e$ is meta-variable)
- $\langle x. x \ y \rangle$: $y$ is free in the term $\langle x. x \ y \rangle$
- $x$ is bound in the term $\langle x. x \ y \rangle$

$\lambda$-calculus uses static binding
**Free Variables**

- $\text{free} (x) = x$
- $\text{free}(e_1 e_2) = \text{free}(e_1) \cup \text{free}(e_2)$
- $\text{free} (\backslash x.e) = \text{free}(e) - \{x\}$
**Free Variables (Inductive Rules)**

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(e_1) &= S_1 \quad FV(e_2) = S_2 \\
FV(e_1 \cdot e_2) &= S_1 \cup S_2 \\
FV(e) &= S \\
FV(\lambda x.e) &= S - \{x\}
\end{align*}
\]
**All Variables**

\[ \text{Vars}(x) = \{x\} \]

\[ \text{Vars}(e_1, e_2) = \text{Vars}(e_1) \cup \text{Vars}(e_2) \]

\[ \text{Vars}(\backslash x. e) = \text{Vars}(e) \cup \{x\} \]
**Substitution**

- \( e[v/x] \) is the term in which all *free* occurrences of \( x \) in \( e \) are replaced with \( v \).
- this replacement operation is called *substitution*.

\[
\begin{align*}
(\\lambda x.\ \lambda y.\ \lambda z\ z)[\lambda w.\ w/z] &= \\lambda x.\ \lambda y.\ (\\lambda w.\ w)\ (\\lambda w.\ w) \\
(\\lambda x.\ \lambda z.\ z\ z)[\lambda w.\ w/z] &= \\lambda x.\ \lambda z.\ z\ z \\
(\\lambda x.\ x\ z)[x/z] &= \\lambda x.\ x\ x \\
(\\lambda x.\ x\ z)[x/z] &= (\\lambda y.\ y\ z)[x/z] = \\lambda y.\ y\ x
\end{align*}
\]

alpha-equivalent expressions = the same except for consistent renaming of bound variables.
“Special” substitution (ignoring capture issues)

Definition of $e_1[[e/x]]$ assuming $\text{FV}(e) \cap \text{Vars}(e_1) = \{\}$:

$x[[e/x]] = e$
$y[[e/x]] = y \quad \text{(if } y \neq x\text{)}$
$e_1 e_2[[e/x]] = (e_1[[e/x]]) (e_2[[e/x]])$
$(\text{\textla} x. e_1) [[e/x]] = \text{\textla} x. e_1$
$(\text{\textla} y. e_1) [[e/x]] = \text{\textla} y. (e_1[[e/x]]) \quad \text{(if } y \neq x\text{)}$
**Alpha-Equivalence**

In order to avoid variable clashes, it is very convenient to **alpha-convert** expressions so that **bound variables** don’t get in the way.

**eg:** to alpha-convert \( \lambda x.e \) we:

1. pick \( z \) such that \( z \) not in \( \text{Vars}(\lambda x.e) \)
2. return \( \lambda z.(e[[z/x]]) \)

We previously defined \( e[[z/x]] \) in such a way that it is a total function when \( z \) is not in \( \text{Vars}(\lambda x.e) \)

**Terminology:** Expressions \( e_1 \) and \( e_2 \) are called **alpha-equivalent** when they are the same after alpha-converting some of their bound variables.
CAPTURE-AVOIDING SUBSTITUTION

Defined inductively on the structure of expression $e$:

\[
\begin{align*}
  x [e/x] &= e \\
  y [e/x] &= y \quad \text{(if } y \neq x) \\
  e_1 e_2 [e/x] &= (e_1 [e/x]) (e_2 [e/x]) \\
  \backslash x.e_1 [e/x] &= \backslash x.e_1 \\
  \backslash y.e_1 [e/x] &= \backslash y.(e_1 [e/x]) \quad \text{(if } y \neq x \text{ and } y \not\in FV(e)) \\
  \backslash y.e_1 [e/x] &= \backslash z.((e_1[\{z/y\}] [e/x]) \quad \text{(if } y \neq x \text{ and } y \in FV(e)) \\
  &\text{for some } z \text{ such that} \\
  z &\not\in FV(e) \cup Vars(e_1)
\end{align*}
\]
OPERATIONAL SEMANTICS

- single-step evaluation (judgment form): $e \rightarrow e'$

- primary rule (beta reduction):
  $$\text{(\textcolor{red}{\textbf{\textbackslash x.e1}} \ e2) \rightarrow e1 [e2/x]}$$

- A term of the form $(\textcolor{red}{\textbf{\textbackslash x.e1}}) \ e2$ is called redex (reducible expression).
**Evaluation Strategies**

- let id = \x. x, consider following exp with 3 redexes:
  
  \[ \text{id (id (\z. \text{id z}))} \]
  
  \[ \text{id (id (\z. \text{id z}))} \]
  
  \[ \text{id (id (\z. \text{id z}))} \]

- Each strategy defines which redex in an expression gets reduced (fired) on the next step of evaluation

- *Full beta-reduction*: any redex
  
  \[ \text{id (id (\z. \text{id z}))} \]
  
  \[ \rightarrow \text{id (id (\z. z))} \]
  
  \[ \rightarrow \text{id (\z. z)} \]
  
  \[ \rightarrow \text{\z. z} \]
EVALUATION STRATEGIES

- **Normal order**: leftmost, outermost redex first
  
  \[ \text{id} \left( \text{id} \left( \lambda z. \text{id} \; z \right) \right) \]
  
  \[ \rightarrow \; \text{id} \left( \lambda z. \text{id} \; z \right) \]
  
  \[ \rightarrow \; \lambda z. \text{id} \; z \]
  
  \[ \rightarrow \; \lambda z. \; z \]

- **Call-by-name**: similar to normal order except NO reduction inside lambda abstractions
  
  \[ \text{id} \left( \text{id} \left( \lambda z. \text{id} \; z \right) \right) \]
  
  \[ \rightarrow \; \text{id} \left( \lambda z. \text{id} \; z \right) \]
  
  \[ \rightarrow \; \lambda z. \text{id} \; z \]
  
  \[ \rightarrow \; \lambda z. \; z \]
EVALUATION STRATEGIES

- *Call-by-value*: only outermost redex, whose RHS must be a value, no reduction inside abstraction
  - values are \[ v ::= \lambda x.e \] (lambda abstractions)
    
    \[
    \begin{align*}
    \text{id } (\text{id } (\lambda z. \text{id } z)) & \rightarrow \text{id } (\lambda z. \text{id } z) \\
    & \rightarrow \lambda z. \text{id } z
    \end{align*}
    \]
**Another Example (Diff between call by name and call by value)**

- **Call by name:**
  \[
  (\lambda x. y) ((\lambda x. x x) (\lambda x. x x))
  \]
  \[
  \rightarrow y
  \]

- **Call by value:**
  \[
  (\lambda x. y) ((\lambda x. x x) (\lambda x. x x))
  \]
  \[
  \rightarrow (\lambda x. y) ((\lambda x. x x) (\lambda x. x x))
  \]
  \[
  \rightarrow (\lambda x. y) ((\lambda x. x x) (\lambda x. x x))
  \]
  \[
  \rightarrow (\lambda x. y) ((\lambda x. x x) (\lambda x. x x))
  \]
  \[
  \rightarrow \ldots
  \]
CALL-BY-VALUE OPERATIONAL SEMANTICS

- Basic rule

\[ (\lambda x.e) \, v \rightarrow e \mid [v/x] \]

- Search rules:

\[
\begin{align*}
\frac{e_1 \rightarrow e_1'}{e_1 \, e_2 \rightarrow e_1' \, e_2} & \quad \frac{e_2 \rightarrow e_2'}{v \, e_2 \rightarrow v \, e_2'}
\end{align*}
\]

- Notice, evaluation is left to right
ALTERNATIVES

\[(\lambda x. e) \, v \rightarrow e \mid [v/x]\]

\[e_1 \rightarrow e_1'\]
\[e_1 \, e_2 \rightarrow e_1' \, e_2\]

\[e_2 \rightarrow e_2'\]
\[v \, e_2 \rightarrow v \, e_2'\]

call-by-value

call-by-name
ALTERNATIVES

\[(\lambda x.e) \, v \rightarrow e \ [v/x]\]

\[e_1 \rightarrow e_1'\]
\[e_1 \, e_2 \rightarrow e_1' \, e_2\]

\[e_2 \rightarrow e_2'\]
\[\nu \, e_2 \rightarrow \nu \, e_2'\]

\[
\text{call-by-value}
\]

\[(\lambda x.e_1) \, e_2 \rightarrow e_1 \ [e_2/x]\]

\[e_1 \rightarrow e_1'\]
\[e_1 \, e_2 \rightarrow e_1' \, e_2\]

\[
\text{normal order}
\]
ALTERNATIVES

\[(\lambda x. e) v \rightarrow e [v/x]\]

\[e_1 \rightarrow e_1'\]
\[e_1 e_2 \rightarrow e_1' e_2\]

\[v e_2 \rightarrow v e_2'\]
call-by-value

\[(\lambda x. e_1) e_2 \rightarrow e_1 [e_2/x]\]

\[e_1 \rightarrow e_1'\]
\[e_1 e_2 \rightarrow e_1' e_2\]

\[e_2 \rightarrow e_2'\]
\[e_1 e_2 \rightarrow e_1 e_2'\]

\[\lambda x. e \rightarrow \lambda x. e'\]
call-by-value

\[e \rightarrow e'\]
\[\lambda e \rightarrow \lambda e'\]
full beta-reduction
ALTERNATIVES

(\x.e) v \rightarrow e [v/x]

e1 \rightarrow e1'

\frac{e1 e2}{e1 e2' e2}

\frac{e2}{v e2' v e2'}

call-by-value

(\x.e) v \rightarrow e [v/x]

\frac{e1}{e1'

\frac{e1 v}{e1' v}

\frac{v e2}{v e2'}

\frac{e1 e2}{e1 e2' e2'}

right-to-left call-by-value
PROVING THEOREMS ABOUT O.S.

Call-by-value o.s.:

\[
\begin{align*}
(\lambda x.e) \; v \rightarrow e \; [v/\!x] \\
\frac{e_1 \rightarrow e_1'}{e_1 \; e_2 \rightarrow e_1' \; e_2} \\
\frac{e_2 \rightarrow e_2'}{v \; e_2 \rightarrow v \; e_2'}
\end{align*}
\]

To prove property \(P\) of \(e_1 \rightarrow e_2\), there are 3 cases:

- **Case:**
  \[
  (\lambda x.e) \; v \rightarrow e \; [v/\!x]
  \]
  **Must prove:** \(P((\lambda x.e) \; v \rightarrow e \; [v/\!x])\)
  **Often requires a related property of substitution \(e \; [v/\!x]\)**

- **Case:**
  \[
  e_1 \rightarrow e_1'
  \]
  \[
  e_1 \; e_2 \rightarrow e_1' \; e_2
  \]
  **IH = \(P(e_1 \rightarrow e_1')\)**
  **Must prove:** \(P(e_1 \; e_2 \rightarrow e_1' \; e_2)\)

- **Case:**
  \[
  e_2 \rightarrow e_2'
  \]
  \[
  v \; e_2 \rightarrow v \; e_2'
  \]
  **IH = \(P(e_2 \rightarrow e_2')\)**
  **Must prove:** \(P(v \; e_2 \rightarrow v \; e_2')\)
MULTI-STEP OP. SEMANTICS

- Given a single step op sem. relation:
  \[ e_1 \rightarrow e_2 \]

- We extend it to a multi-step relation by taking its “reflexive, transitive closure:”

\[
\begin{align*}
  &e_1 \rightarrow \star e_1 \quad \text{(reflexivity)} & e_1 \rightarrow e_2 & e_2 \rightarrow \star e_3 \quad \text{(transitivity)} \\
  &\text{or} & \quad e_1 \rightarrow \star e_3
\end{align*}
\]
PROVING THEOREMS ABOUT O.S.

Call-by-value o.s.:

\[
\begin{align*}
\text{(reflexivity)} & & e_1 \rightarrow^* e_1 \\
\text{(transitivity)} & & e_1 \rightarrow e_2 & e_2 \rightarrow^* e_3 \\
& & e_1 \rightarrow^* e_3
\end{align*}
\]

To prove property $P$ of $e_1 \rightarrow^* e_2$, given you’ve already proven property $P’$ of $e_1 \rightarrow e_2$, there are 2 cases:

**case:**

\[
\begin{align*}
e_1 \rightarrow^* e_1
\end{align*}
\]

Must prove: $P(e_1 \rightarrow^* e_1)$ directly

**case:**

\[
\begin{align*}
e_1 \rightarrow e_2 & e_2 \rightarrow^* e_3 \\
& e_1 \rightarrow^* e_3 \quad \text{IH} = P(e_2 \rightarrow^* e_3)
\end{align*}
\]

Also available: $P’(e_1 \rightarrow e_2)$

Must prove: $P(e_1 \rightarrow^* e_3)$
**Example**

Definition: An expression $e$ is **closed** if $\text{FV}(e) = \{ \}$. 

Theorem: 
If $e_1$ is closed and $e_1 \rightarrow^* e_2$ then $e_2$ is closed.  
Proof: by induction on derivation of $e_1 \rightarrow^* e_2$. 

(We need to prove lemma: if $e_1$ is closed and $e_1 \rightarrow e_2$, then $e_2$ is closed.)