Today, we will learn about

- Lambda calculus
- Full $\beta$-reduction
- Call-by-value semantics
- Call-by-name semantics
Lambda calculus: Intuition

A function is a rule for determining a value from an argument. Some examples of functions in mathematics are

\[ f(x) = x^3 \]
\[ g(y) = y^3 - 2y^2 + 5y - 6. \]
Pure vs Applied Lambda Calculus

- The pure $\lambda$-calculus contains just function definitions (called *abstractions*), variables, and function *applications*.

- If we add additional data types and operations (such as integers and addition), we have an *applied* $\lambda$-calculus.
Pure Lambda Calculus: Syntax

\[ e ::= \begin{array}{l}
\text{x} \quad \text{variable} \\
\mid \lambda x. \ e \quad \text{abstraction} \\
\mid e_1 \ e_2 \quad \text{application}
\end{array} \]
Abstractions
Abstractions

- An abstraction $\lambda x. e$ is a function
- Variable $x$ is the *argument*
- Expression $e$ is the *body* of the function.
- The expression $\lambda y. y \times y$ is a function that takes an argument $y$ and returns square of $y$. 
Applications

- An application $e_1 e_2$ requires that $e_1$ is (or evaluates to) a function, and then applies the function to the expression $e_2$.

- For example, $(\lambda y. y \times y) \ 5$ is 25
Examples

\( \lambda x. x \) a lambda abstraction called the *identity function*

\( \lambda x. (f (g x)) \) another abstraction

\( (\lambda x. x) 42 \) an application

\( \lambda y. \lambda x. x \) an abstraction, ignores its argument and returns the identity function
Lambda expressions extend as far to the right as possible

\[ \lambda x. x \quad \lambda y. y \text{ is the same as } \lambda x. (x \ (\lambda y. y)), \text{ and is not the same as } (\lambda x. x) \ (\lambda y. y). \]
Application is left-associative

\[ e_1 \ e_2 \ e_3 \text{ is the same as } (e_1 \ e_2) \ e_3. \]
Use parentheses!

In general, use parentheses to make the parsing of a lambda expression clear if you are in doubt.
Variable binding

▶ An occurrence of a variable \( x \) in a term is bound if there is an enclosing \( \lambda x. e \); otherwise, it is free.

▶ A closed term is one in which all identifiers are bound.
Variable binding: \( \lambda x. (x (\lambda y. y a) x) y \)
Variable binding: $\lambda x. (x (\lambda y. y \ a) \ x) \ y$

- Both occurrences of $x$ are bound
- The first occurrence of $y$ is bound
- The $a$ is free
- The last $y$ is also free, since it is outside the scope of the $\lambda y$. 
The symbol $\lambda$ is a binding operator: variable $x$ is bound in $e$ in the expression $\lambda x. e$. 
\(\alpha\)-equivalence

- \(\lambda x. x\) is the same function as \(\lambda y. y\).

- Expressions \(e_1\) and \(e_2\) that differ only in the name of bound variables are called \(\alpha\)-equivalent ("alpha equivalent").

- Sometimes written \(e_1 =_\alpha e_2\).
Quiz: $\alpha$-equivalence

Are $\lambda x. \lambda y. x \ y$ and $\lambda y. \lambda x. y \ x$ $\alpha$-equivalent?
Higher-order functions

- In lambda calculus, functions are values.

- In the pure lambda calculus, every value is a function, and every result is a function!
Higher-order functions

$$\lambda f. f\;42$$
Higher-order functions

\[ \lambda v. \lambda f. (f \ v) \]

Takes an argument \( v \) and returns a function that applies its own argument (a function) to \( v \).
Semantics
We would like to regard \((\lambda x. e_1) \; e_2\) as equivalent to \(e_1\) where every (free) occurrence of \(x\) is replaced with \(e_2\).

E.g. we would like to regard \((\lambda y. y \times y) \; 5\) as equivalent to \(5 \times 5\).
We write $e_1\{e_2/x\}$ to mean expression $e_1$ with all free occurrences of $x$ replaced with $e_2$.

We call $(\lambda x. e_1) \ e_2$ and $e_1\{e_2/x\}$ $\beta$-equivalent.

Rewriting $(\lambda x. e_1) \ e_2$ into $e_1\{e_2/x\}$ is called a $\beta$-reduction.

This corresponds to executing a lambda calculus expression.
Different semantics for the lambda calculus

\[(\lambda x. x + x) \ ((\lambda y. y) \ 5)\]
Different semantics for the lambda calculus

\[(\lambda x. x + x) ((\lambda y. y) \ 5)\]

We could use \(\beta\)-reduction to get either \(((\lambda y. y) \ 5) + ((\lambda y. y) \ 5)\) or \((\lambda x. x + x) \ 5\).
Evaluation strategies: Full $\beta$-reduction

$$(\lambda x. e_1)\ e_2 \text{ to step to } e_1\{e_2/x\} \text{ at any time.}$$
Full $\beta$-reduction: small-step operational semantics

$$\frac{e_1 \rightarrow e_1'}{e_1 \ e_2 \rightarrow e_1' \ e_2} \quad \frac{e_2 \rightarrow e_2'}{e_1 \ e_2 \rightarrow e_1 \ e_2'}$$

$$\frac{e \rightarrow e'}{\lambda x. \ e \rightarrow \lambda x. \ e'}$$

$\beta$-REDUCTION $$(\lambda x. \ e_1) \ e_2 \rightarrow e_1\{e_2/x\}$$
Normal form

A term $e$ is said to be in *normal form* when there is no $e'$ such that $e \rightarrow e'$. 
Not every term has a normal form under full $\beta$-reduction.

Consider $\Omega = (\lambda x. x \ x) \ (\lambda x. x \ x)$.

$\Omega = (\lambda x. x \ x) \ (\lambda x. x \ x) \rightarrow (\lambda x. x \ x) \ (\lambda x. x \ x) = \Omega$

It’s an infinite loop!
Well-behaved nondeterminism

$$(\lambda x. \lambda y. y) \Omega (\lambda z. z)$$
Well-behaved nondeterminism

\[(\lambda x. \lambda y. y) \Omega (\lambda z. z)\]

This term has two redexes in it, the one with abstraction \(\lambda x\), and the one inside \(\Omega\).
Well-behaved nondeterminism

- The full $\beta$-reduction strategy is non-deterministic.

- When a term has a normal form, however, it never has more than one.
Theorem (Confluence)

If $e \longrightarrow^* e_1$ and $e \longrightarrow^* e_2$ then there exists $e'$ such that $e_1 \longrightarrow^* e'$ and $e_2 \longrightarrow^* e'$. 
Full $\beta$-reduction is confluent

Corollary

If $e \rightarrow^* e_1$ and $e \rightarrow^* e_2$ and both $e_1$ and $e_2$ are in normal form, then $e_1 = e_2$.

Proof.
An easy consequence of confluence.
Normal Order Evaluation

- *Normal order evaluation* uses the full $\beta$-reduction rules, except the left-most redex is always reduced first.

- Will eventually yield the normal form, if one exists.

- Allows reducing redexes inside abstractions
Call-by-value

- Call-by-value only allows an application to reduce after its argument has been reduced to a value and does not allow evaluation under a λ.

- Given an application \((\lambda x. e_1) \, e_2\), CBV semantics makes sure that \(e_2\) is a value before calling the function.

- A value is an expression that can not be reduced/executed/simplified any further.
CBV: Small step operational semantics

\[ e_1 \rightarrow e'_1 \]

\[ e_1 \ e_2 \rightarrow e'_1 \ e_2 \]

\[ e \rightarrow e' \]

\[ v \ e \rightarrow v \ e' \]

\[ \beta\text{-REDUCTION} \]

\[ (\lambda x. \ e) \ v \rightarrow e\{v/x\} \]
CBV: Examples

\[(\lambda x. \lambda y. y \; x) \; (5 + 2) \; \lambda x. x + 1 \quad \rightarrow \quad (\lambda x. \lambda y. y \; x) \; 7 \; \lambda x. x + 1 \]
\[\rightarrow (\lambda y. y \; 7) \; \lambda x. x + 1 \]
\[\rightarrow (\lambda x. x + 1) \; 7 \]
\[\rightarrow 7 + 1 \]
\[\rightarrow 8 \]
\[(\lambda f. f\ 7)\ (\lambda x. x\ x)\ \lambda y.\ y) \rightarrow (\lambda f. f\ 7)\ ((\lambda y.\ y)\ (\lambda y.\ y))
\rightarrow (\lambda f. f\ 7)\ (\lambda y.\ y)
\rightarrow (\lambda y.\ y)\ 7
\rightarrow 7\]
Call-by-name semantics

- More permissive than CBV.
- Less permissive than full $\beta$-reduction.
- Applies the function as soon as possible.
- No need to ensure that the expression to which a function is applied is a value.
Call-by-name semantics

$$\begin{align*}
e_1 & \rightarrow e'_1 \\
e_1 & \rightarrow e'_1 e_2 \\
\beta\text{-REDUCTION} & \frac{(\lambda x. e_1) \; e_2}{e_1 \{ e_2 / x \}}
\end{align*}$$
Call-by-name semantics: example

\((\lambda x. \lambda y. y \ x) \ (5 + 2) \ \lambda x. x + 1\) \[\rightarrow\] \((\lambda y. y \ (5 + 2)) \ \lambda x. x + 1\)
\[\rightarrow\] \((\lambda x. x + 1) \ (5 + 2)\)
\[\rightarrow\] \((5 + 2) + 1\)
\[\rightarrow\] \(7 + 1\)
\[\rightarrow\] \(8\)

compare to CBV:

\((\lambda x. \lambda y. y \ x) \ (5 + 2) \ \lambda x. x + 1\) \[\rightarrow\] \((\lambda x. \lambda y. y \ x) \ 7 \ \lambda x. x + 1\)
\[\rightarrow\] \((\lambda y. y \ 7) \ \lambda x. x + 1\)
\[\rightarrow\] \((\lambda x. x + 1) \ 7\)
\[\rightarrow\] \(7 + 1\)
\[\rightarrow\] \(8\)
Call-by-name semantics: example

\[(\lambda f. f \ 7) \ (\ (\lambda x. x \ x) \ \lambda y. y) \rightarrow ((\lambda x. x \ x) \ \lambda y. y) \ 7\]
\[\rightarrow ((\lambda y. y) \ (\lambda y. y)) \ 7\]
\[\rightarrow (\lambda y. y) \ 7\]
\[\rightarrow 7\]

compare to CBV:

\[(\lambda f. f \ 7) \ (\ (\lambda x. x \ x) \ \lambda y. y) \rightarrow (\lambda f. f \ 7) \ ((\lambda y. y) \ (\lambda y. y))\]
\[\rightarrow (\lambda f. f \ 7) \ (\lambda y. y)\]
\[\rightarrow (\lambda y. y) \ 7\]
\[\rightarrow 7\]
CBV vs CBN

One way in which CBV and CBN differ is when arguments to functions have no normal forms.

\[(\lambda x. (\lambda y. y)) \Omega\]

Under CBV semantics, this term does not have a normal form.

If we use CBN semantics, then we have

\[(\lambda x. (\lambda y. y)) \Omega \rightarrow_{\text{CBN}} \lambda y. y\]
CBV and CBN

- CBV and CBN are common evaluation orders
- Many programming languages use CBV semantics
- “Lazy” languages, such as Haskell, typically use CBN semantics, a more efficient semantics similar to CBN in that it does not evaluate actual arguments unless necessary
- However, Call-by-value semantics ensures that arguments are evaluated at most once.
If possible, give a program that cannot reduce in CBN and CBV, but reduces in full $\beta$-reduction.

If possible, give a program that steps to the same expression in CBN and CBV.

Formulate the rules of CBV in big-step style.

How would you create a let-binding in lambda calculus?

How do we define $e_1\{e_2/x\}$ formally?
Scratchpad
CBV in big-step

\[
\lambda x. e \Downarrow \lambda x. e \\
\]

\[
e_1 \Downarrow \lambda x. e_{12} \quad e_2 \Downarrow v_2 \quad e_{12}\{v_2/x\} \Downarrow e' \text{ } \\
\]

\[
e_1 \quad e_2 \Downarrow e' \quad 
\]
\[ e_1 \{ e_2 / x \} \] formally

\[
x\{e/x\} = e
\]
\[
y\{e/x\} = y \quad \text{if } y \neq x
\]
\[
(\lambda y. \, e_1)\{e/x\} = \lambda y. (e_1\{e/x\}) \quad \text{if } y \neq x \text{ and } y \notin FV(e)
\]
\[
(e_1 \, e_2)\{e/x\} = (e_1\{e/x\}) \, (e_2\{e/x\})
\]