Lecture 6
The λ-calculus
Instructor: Hossein Hojjat

September 11, 2017
Many features in programming languages are not essential: they are for convenience

- Multi-argument functions
- Use Currying

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>def</td>
<td>f(x, y, z)</td>
</tr>
</tbody>
</table>

- Loops
- Use recursion

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>while</td>
<td>(x == y)</td>
</tr>
<tr>
<td></td>
<td>{...}</td>
</tr>
</tbody>
</table>

- Side effects
- Use functional programming

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>(x = 1)</td>
</tr>
</tbody>
</table>

- What languages features are really necessary?
Core Language

• Goal: Come up with a “core” language
  - As small as possible
  - Still Turing complete

• Gives an opportunity of studying important language features
Core Language

- Goal: Come up with a “core” language
  - As small as possible
  - Still Turing complete
- Gives an opportunity of studying important language features

\(\lambda\)-calculus

- Peter J Landin (1960s):
  \(\lambda\)-calculus can be used to model a complex programming language
- A programming language is nothing more than \(\lambda\)-calculus plus some syntactic sugar
  - “A Correspondence Between ALGOL 60 and Church’s Lambda-Notation”(1965)
  - “The Next 700 Programming Languages”(1966)
- \(\lambda\)-calculus plays a similar role for PL research as Turing machines do for computability
• A $\lambda$-calculus expression is defined as

$$e ::= x \quad \text{variable}$$

$$| \lambda x . e \quad \text{abstraction}$$

$$| e \; e \quad \text{application}$$

**Application**

$$e_1 \; e_2$$

- function
- argument

• Application is left associative

$$(e_1 \; e_2 \; e_3 \; e_4) \equiv (((e_1 \; e_2) \; e_3) \; e_4)$$

**Abstraction**

$$\lambda x . e$$

- binder
- body

• Scope of the dot in an abstraction extends as far to the right as possible

$$\lambda x . e_1 \; e_2 \; e_3 \quad \text{means} \quad (\lambda x . (e_1 \; e_2 \; e_3)) \quad \text{and not} \quad ((\lambda x . e_1 \; e_2) \; e_3)$$
Examples of Lambda Expressions

- The identity function
Examples of Lambda Expressions

- The identity function

\[ \lambda x.x \]
Examples of Lambda Expressions

- The identity function
  \[ \lambda x . x \]

- Function that takes two other functions and produces their composition
Examples of Lambda Expressions

- The identity function

\[ \lambda x.x \]

- Function that takes two other functions and produces their composition

\[ \lambda f.\lambda g.(f \ g) \]
Examples of Lambda Expressions

- The identity function

  \[ \lambda x. x \]

- Function that takes two other functions and produces their composition

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

- Function that flips the order of two arguments that are passed to a function
Examples of Lambda Expressions

- The identity function
  \[ \lambda x. x \]

- Function that takes two other functions and produces their composition
  \[ \lambda f. \lambda g. (f \ g) \]

- Function that flips the order of two arguments that are passed to a function
  \[ \lambda f. \lambda x. \lambda y. (f \ y \ x) \]
Multiple Arguments

- $\lambda$-calculus provides no built-in support for multi-argument functions.
- Do we need an extension $\lambda(x_1 \cdots x_n)\ e$ to support multiple arguments?
Multiple Arguments

- \( \lambda \)-calculus provides no built-in support for multi-argument functions
- Do we need an extension \( \lambda(x_1 \cdots x_n) \, e \) to support multiple arguments?
- **Currying**: reducing a function with multiple arguments to functions with a single argument

\[
(\lambda(x_1 \cdots x_n).e) \Rightarrow (\lambda x_1. (\lambda \cdots (\lambda x_n.e) \cdots))
\]

Example.

- Consider the function \( \lambda(x, y). (x \, y) \)
- Curried form: \( f = \lambda x. \lambda y. (x \, y) \)
- \( g = (f \, \lambda x. x) \) is a function that takes a single argument and applies identity to it
Static Scoping

• Scope of a variable: portion of program where the identifier is accessible

• An abstraction $\lambda x.e$ binds the variable $x$ in $e$
  • $e$ is the scope of $x$
  • $x$ is bound in $\lambda x.e$

• $\lambda$-calculus uses static scoping
  • Scope of variable is fixed at compile-time to the smallest block containing the variable declaration

• $(\lambda x.x\ (\lambda x.x))\ z$
  • Rightmost $x$ refers to the second binding
  • Takes its argument and applies it to the identity function
Free and Bound Variables

- Occurrences of $x$ in the body $e$ are bound
- Nonbound variable occurrences are called free

**Free Variables**

- $\text{FV}(x) = \{x\}$
- $\text{FV}(e_1 e_2) = \text{FV}(e_1) \cup \text{FV}(e_2)$
- $\text{FV}(\lambda x. e) = \text{FV}(e) - \{x\}$
Free and Bound Variables

\[ \lambda x. (x \ (\lambda y. y \ z) \ x) \ y \]
Free and Bound Variables

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

bound
Free and Bound Variables

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

\[ \text{bound} \quad \text{bound} \]
A $\lambda$-expression without free variables is **closed** or a **combinator**.

An expression is **open** if it is not closed.

Example. Some interesting combinators (we will discuss these combinators more)

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

\[ Y = \lambda f. (\lambda x. f\ (x\ x)) (\lambda x. f\ (x\ x)) \]
Free and Bound Variables

\[ \lambda x. \left( x \left( \lambda y. y \ z \right) \ x \right) \ y \]

- A \lambda\text{-expression without free variables is closed or a combinator}
- An expression is open if it is not closed

Example. Some interesting combinators (we will discuss these combinators more)

\[ \Omega = \left( \lambda x. x \ x \right) \left( \lambda x. x \ x \right) \]

\[ Y = \lambda f. \left( \lambda x. f \left( x \ x \right) \right) \left( \lambda x. f \left( x \ x \right) \right) \]
Free and Bound Variables

- A $\lambda$-expression without free variables is **closed** or a **combinator**
- An expression is **open** if it is not closed

\[
\lambda x. \left( x \, (\lambda y.y \, z) \, x \right) \, y
\]

Bound: $x$, $y$, $z$
Boundfree: $\lambda y.y$
Free: $x$
Free and Bound Variables

\[ \lambda x. \left( x \ (\lambda y. y \ z) \ x \right) \ y \]

bound  boundfree  bound  free

- A λ-expression without free variables is **closed** or a **combinator**
- An expression is **open** if it is not closed

**Example.**

Some interesting combinators (we will discuss these combinators more)

\[ \Omega = (\lambda x. x \ x)(\lambda x. x \ x) \]
\[ Y = \lambda f.(\lambda x. f(x \ x)) \ (\lambda x. f(x \ x)) \]
Free and Bound Variables

- Like in any language with statically nested scoping, we need to worry about variable shadowing.
- Shadowing: a variable declared within a scope has the same name as a variable declared in an outer scope.

\[
\lambda x. x (\lambda x.x) x
\]
Renaming Bound Variables

- **α-equivalence**: λ-expressions that can be obtained from one another by renaming the bound variables are identical

- **Example**: \( \lambda x.x \) is identical to \( \lambda y.y \) and to \( \lambda z.z \)

- **Intuition**: Change the name of a formal argument and of all its occurrences in the function body: function behavior does not change

- **α-conversion**: renaming the bound variables in the expression

- Always try to rename bound variables so that they are all unique
  - Write \( \lambda x.x (\lambda y.y) x \) instead of \( \lambda x.x (\lambda x.x) x \)
  - Makes it easy to see the scope of bindings
Substitution

$e[x \mapsto e_1]$

\(\lambda\)-expression obtained by replacing each free occurrence of the variable \(x\) in \(e\) by the \(\lambda\)-expression \(e_1\)

- **Valid** Substitution: no free variable in \(e_1\) becomes bound as a result of the substitution \(e[x \mapsto e_1]\)
- **Invalid** Substitution: involves a variable capture or a name clash

Example:
- an invalid substitution:
  \[(\lambda x . x \ y)[y \mapsto x] \neq \lambda x . x \ x\]
- first function: applies argument to \(y\)
- second function: identity
Substitution

- Substitution is defined by cases on $e[x \mapsto e']$

**Variable**

$$y[x \mapsto e'] = e' \quad \text{if } x = y$$
$$y[x \mapsto e'] = y \quad \text{otherwise}$$

**Application**

$$(e_1 e_2)[x \mapsto e'] = (e_1[x \mapsto e'] e_2[x \mapsto e'])$$

**Abstraction**

$$(\lambda y.e_1)[x \mapsto e'] = \lambda y.e_1 \quad \text{if } x = y$$
$$(\lambda y.e_1)[x \mapsto e'] = \lambda z.((e_1[y \mapsto z])[x \mapsto e']) \quad \text{otherwise}$$

where $z \notin (\text{FV}(e_1) \cup \text{FV}(e') \cup \{x\})$
• In λ-calculus there is one computation rule called β-reduction

\[(\lambda x.e) \ e_1 \rightarrow_\beta \ e[x \mapsto e_1]\]

• A β-redex (reducible expression) is a term of form \((\lambda x.e) \ e_1\)

• λ-definable functions coincide with the Turing-computable functions (λ-calculus is Turing-complete)

• Any possible computation can be defined in terms of β-reduction rule
\[\beta\text{-reduction Exercise}\]

- Apply the \(\beta\)-reductions in the following expression

\[((\lambda x.\lambda y.x) y) z\]
• Apply the $\beta$-reductions in the following expression

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

$\rightarrow_{\beta} \quad ((\lambda y.x)[x \mapsto y])\ z \quad$ substitute $y$ for $x$ in the body of $\lambda y.x$

$\rightarrow_{\alpha} \quad ((\lambda y'.x)[x \mapsto y])\ z \quad$ after $\alpha$-conversion

$\rightarrow \quad (\lambda y'.y)\ z \quad$ first $\beta$-reduction complete

$\rightarrow_{\beta} \quad y[y' \mapsto z] \quad$ substitute $z$ for $y'$ in $y$

$\rightarrow \quad y \quad$ second $\beta$-reduction complete
β-reduction Exercise

• Apply the β-reductions in the following expression

\[ \Omega = (\lambda x. x \ x)(\lambda x. x \ x) \]
• Apply the $\beta$-reductions in the following expression

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

• Each time you beta-reduce you get the same expression back

• $\lambda$-calculus is equivalent to Turing machine

• Turing machine may fail to halt
- \(\eta\)-reduction is useful to eliminate redundant \(\lambda\)-abstractions
  \[ \lambda x. (e x) \rightarrow_{\eta} e \quad \text{if} \ x \notin \text{FV}(e) \]

- Motivation for \(\eta\)-conversion:
- \(\lambda x. (e x)\) and \(e\) behave identically as functions
  \[ (\lambda x. (e x)) \ u \rightarrow_{\beta} (e \ u) \quad \text{if} \ x \notin \text{FV}(e) \]

Example:

\[
\begin{align*}
(\lambda x. (\lambda y.x y) (y \ w)) \\
\rightarrow_{\alpha} (\lambda x. (\lambda y'.x y') (y \ w)) \\
\rightarrow_{\beta} \lambda y'.((y \ w) y') \\
\rightarrow_{\eta} (y \ w)
\end{align*}
\]
Computing with $\lambda$-calculus

- $\lambda$-calculus is a **core** programming language: it is Turing complete
- How can we possibly compute with the $\lambda$-calculus when we have no data to manipulate?
  1. no numbers
  2. no data-structures
  3. no control structures (if-then-else, loops)
Computing with $\lambda$-calculus

- $\lambda$-calculus is a **core** programming language: it is Turing complete
- How can we possibly compute with the $\lambda$-calculus when we have no data to manipulate?
  1. no numbers
  2. no data-structures
  3. no control structures (if-then-else, loops)

**Church Encoding:**

Representation of data and operators in the lambda calculus

- Not intended as a practical implementation of primitive data types
- It shows that other primitive data types are not required to represent any calculation
Encoding Booleans

- We need to define functions TRUE, FALSE, AND, NOT, IF that behave as expected
- For example:

\[
\begin{align*}
\text{AND TRUE FALSE} &= \text{FALSE} \\
\text{NOT FALSE} &= \text{TRUE} \\
\text{IF TRUE } e_1 e_2 &= e_1 \\
\text{IF FALSE } e_1 e_2 &= e_2
\end{align*}
\]
We can define TRUE and FALSE as e.g. following:

TRUE $\equiv \lambda x.\lambda y.x$

FALSE $\equiv \lambda x.\lambda y.y$

Both TRUE and FALSE take two arguments:
- TRUE returns the first
- FALSE returns the second
• Conditional statement takes three arguments $b, e_t, e_f$ where: $b$ is a Boolean value and $e_t$, $e_f$ are arbitrary $\lambda$-expressions

$$\text{IF} = \lambda b. \lambda e_t. \lambda e_f. \begin{cases} e_t & \text{if } b = \text{true} \\ e_f & \text{if } b = \text{false} \end{cases}$$
• Conditional statement takes three arguments $b, e_t, e_f$ where: $b$ is a Boolean value and $e_t$, $e_f$ are arbitrary $\lambda$-expressions.

$$IF = \lambda b. \lambda e_t. \lambda e_f. \begin{cases} e_t & \text{if } b = \text{true} \\ e_f & \text{if } b = \text{false} \end{cases}$$

• Since $\text{TRUE} \ e_t \ e_f \rightarrow_\beta \ e_t$ and $\text{FALSE} \ e_t \ e_f \rightarrow_\beta \ e_f$ we define

$$IF \triangleq \lambda b. \lambda e_t. \lambda e_f. b \ e_t \ e_f$$
Encoding Booleans

\[
\begin{align*}
\text{NOT} & \triangleq \lambda b. b \text{ FALSE TRUE} \\
\text{AND} & \triangleq \lambda b_1. \lambda b_2. b_1 b_2 \text{ FALSE} \\
\text{OR} & \triangleq \lambda b_1. \lambda b_2. b_1 \text{ TRUE } b_2
\end{align*}
\]
Church Numerals

\begin{align*}
c_0 & \equiv \lambda s. \lambda z. z \\
c_1 & \equiv \lambda s. \lambda z. s \ z \\
c_2 & \equiv \lambda s. \lambda z. s \ (s \ z) \\
c_3 & \equiv \lambda s. \lambda z. s \ (s \ (s \ z)) \\
\cdots
\end{align*}

- Encode numbers with two-argument functions
- “Number \( i \)” composes the first argument \( i \) times, starting with the second argument
- \( z \) stands for “zero” and \( s \) for “successor” (think unary)
Church Numerals

\[
\begin{align*}
  c_0 & \quad \lambda s.\lambda z.z \\
  c_1 & \quad \lambda s.\lambda z.s\ z \\
  c_2 & \quad \lambda s.\lambda z.s\ (s\ z) \\
  c_3 & \quad \lambda s.\lambda z.s\ (s\ (s\ z)) \\
  \cdots
\end{align*}
\]

SUCC $\quad \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$

- successor: take “a number” and return “a number” that (when called) applies $s$ one more time