



CSCI 740 - Programming Language Theory

Lecture 6

The λ -calculus

Instructor: Hossein Hojjat

September 11, 2017

Programming Language Features

Many features in programming languages are not essential:
they are for convenience

<ul style="list-style-type: none">• Multi-argument functions• Use Currying	<pre>def f(x, y, z)</pre>
<ul style="list-style-type: none">• Loops• Use recursion	<pre>while (x == y) { ... }</pre>
<ul style="list-style-type: none">• Side effects• Use functional programming	<pre>var (x = 1)</pre>

- 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

λ -calculus

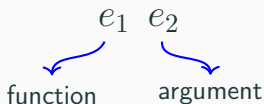
- Peter J Landin (1960s):
 - λ -calculus can be used to model a complex programming language
- A programming language is nothing more than λ -calculus plus some syntactic sugar
 - “A Correspondence Between ALGOL 60 and Church’s Lambda-Notation”(1965)
 - “The Next 700 Programming Languages”(1966)
- λ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

Lambda Expressions

- A λ -calculus expression is defined as

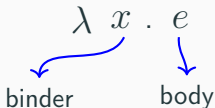
$e ::= x$	variable
$\lambda x.e$	abstraction
$e e$	application

Application



- Application is left associative $(e_1 e_2 e_3 e_4) \equiv (((e_1 e_2) e_3) e_4)$

Abstraction



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

$\lambda x.e_1 e_2 e_3$ means $(\lambda x.(e_1 e_2 e_3))$ and not $((\lambda x.e_1 e_2) e_3)$

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

- λ -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

- λ -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

$$(\lambda(x_1 \cdots x_n).e) \Rightarrow (\lambda x_1.(\lambda \cdots (\lambda x_n.e) \cdots))$$

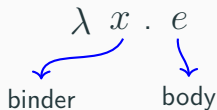
Example.

- 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 $\lambda x.e$
- λ -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

$$\begin{aligned} \text{FV}(x) &= \{x\} \\ \text{FV}(e_1 e_2) &= \text{FV}(e_1) \cup \text{FV}(e_2) \\ \text{FV}(\lambda x.e) &= \text{FV}(e) - \{x\} \end{aligned}$$

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

Free and Bound Variables

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

↑
bound

Free and Bound Variables

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

↑ ↑
bound bound

Free and Bound Variables

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

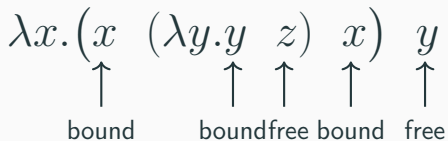
↑ ↑ ↑
bound bound free

Free and Bound Variables

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

↑ ↑ ↑ ↑
bound boundfree bound

Free and Bound Variables



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

Free and Bound Variables

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

↑ ↑ ↑ ↑ ↑
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

$\lambda x. x (\lambda x. x) x$

The diagram illustrates the binding of variables in the lambda expression $\lambda x. x (\lambda x. x) x$. It shows three occurrences of the variable x . The first x is the parameter of the outer lambda abstraction. The second x is the parameter of the inner lambda abstraction. The third x is a free variable. Arrows indicate the binding: a long arrow from the first x to the second x , a shorter arrow from the second x to the third x , and a vertical line from the third x pointing upwards, indicating it is not bound by any lambda.

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

$$e[x \mapsto e_1]$$

λ -expression obtained by replacing each free occurrence of the variable x in e by the λ -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:

$$(\lambda x.x y)[y \mapsto x] \stackrel{?}{=} \lambda x.x x$$

- first function: applies argument to y
- second function: identity

Substitution

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

Variable

$$\begin{aligned}y[x \mapsto e'] &= e' && \text{if } x = y \\y[x \mapsto e'] &= y && \text{otherwise}\end{aligned}$$

Application

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

Abstraction

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

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

β -reduction Exercise

- Apply the β -reductions in the following expression

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

β -reduction Exercise

- Apply the β -reductions in the following expression

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

\rightarrow_{β}	$((\lambda y. x)[x \mapsto y]) z$	substitute y for x in the body of $\lambda y. x$
\rightarrow_{α}	$((\lambda y'. x)[x \mapsto y]) z$	after α -conversion
\rightarrow	$(\lambda y'. y) z$	first β -reduction complete
\rightarrow_{β}	$y[y' \mapsto z]$	substitute z for y' in y
\rightarrow	y	second β -reduction complete

β -reduction Exercise

- Apply the β -reductions in the following expression

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

β -reduction Exercise

- Apply the β -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
- λ -calculus is equivalent to Turing machine
- Turing machine may fail to halt

- η -reduction is useful to eliminate redundant λ -abstractions

$$\lambda x.(e x) \rightarrow_{\eta} e \quad \text{if } x \notin \text{FV}(e)$$

- Motivation for η -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:

$$\begin{aligned} & (\lambda x.(\lambda y.x y) (y w)) \\ \rightarrow_{\alpha} & (\lambda x.(\lambda y'.x y') (y w)) \\ \rightarrow_{\beta} & \lambda y'.((y w) y') \\ \rightarrow_{\eta} & (y w) \end{aligned}$$

Computing with λ -calculus

- λ -calculus is a **core** programming language: it is Turing complete
- How can we possibly compute with the λ -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 λ -calculus

- λ -calculus is a **core** programming language: it is Turing complete
- How can we possibly compute with the λ -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:

AND TRUE FALSE = FALSE

NOT FALSE = TRUE

IF TRUE e_1 e_2 = e_1

IF FALSE e_1 e_2 = e_2

Encoding Booleans

- We can define TRUE and FALSE as e.g. following

$$\text{TRUE} \triangleq \lambda x. \lambda y. x$$

$$\text{FALSE} \triangleq \lambda x. \lambda y. y$$

- Both TRUE and FALSE take two arguments:
 - TRUE returns the first
 - FALSE returns the second

Encoding Booleans

- Conditional statement takes three arguments b, e_t, e_f where:
 b is a Boolean value and e_t, e_f are arbitrary λ -expressions

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

Encoding Booleans

- Conditional statement takes three arguments b, e_t, e_f where: b is a Boolean value and e_t, e_f are arbitrary λ -expressions

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

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

$$\text{IF} \triangleq \lambda b. \lambda e_t. \lambda e_f. b e_t e_f$$

NOT \triangleq $\lambda b.b$ FALSE TRUE

AND \triangleq $\lambda b_1.\lambda b_2.b_1 b_2$ FALSE

OR \triangleq $\lambda b_1.\lambda b_2.b_1$ TRUE b_2

Church Numerals

c_0 $\lambda s.\lambda z.z$
 c_1 $\lambda s.\lambda z.s z$
 c_2 $\lambda s.\lambda z.s (s z)$
 c_3 $\lambda s.\lambda z.s (s (s z))$
...

- 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

c_0 $\lambda s.\lambda z.z$

c_1 $\lambda s.\lambda z.s z$

c_2 $\lambda s.\lambda z.s (s z)$

c_3 $\lambda s.\lambda z.s (s (s z))$

...

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

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