

# Lecture 8 – Lambda Calculus

## COSE212: Programming Languages

Jihyeok Park



2023 Fall

- FVAE – VAE with First-Class Functions
  - First-Class Functions
  - Concrete and Abstract Syntax
  - Interpreter and Natural Semantics with Closures
  - Static and Dynamic Scoping
- In this lecture, we will learn **syntactic sugar** and **lambda calculus**

## 1. Syntactic Sugar

- No More `val`

- F<sub>AE</sub> – Removing `val` from FVAE

- Syntactic Sugar and Desugaring

## 2. Lambda Calculus

- Definition

- Church Encodings

- Church-Turing Thesis

## 1. Syntactic Sugar

- No More `val`

- F<sub>AE</sub> – Removing `val` from FVAE

- Syntactic Sugar and Desugaring

## 2. Lambda Calculus

- Definition

- Church Encodings

- Church-Turing Thesis

```
/* FVAE */  
val x = 1; x + 2
```

It assigns a **value** 1 to the **variable**  $x$ , and then evaluates the **body expression**  $x + 2$  with the environment  $[x \mapsto 1]$ .

It is same as:

```
/* FVAE */  
(x => x + 2)(1)
```

It assigns a **value** (argument) 1 to the **parameter**  $x$ , and then evaluates the **body expression**  $x + 2$  with the environment  $[x \mapsto 1]$ .

In general, the following two expressions are equivalent:

$\text{val } x=e; e'$  is equivalent to  $(\lambda x.e')(e)$

Why?

The following inference rule for the semantics of  $\text{val } x=e; e'$ :

$$\text{VAL} \frac{\sigma \vdash e \Rightarrow v \quad \sigma[x \mapsto v] \vdash e' \Rightarrow v'}{\sigma \vdash \text{val } x = e; e' \Rightarrow v'}$$

is equivalent to the following inference rule for the semantics of  $(\lambda x.e')(e)$ :

$$\text{App} \frac{\text{Fun} \frac{}{\sigma \vdash \lambda x.e' \Rightarrow \langle \lambda x.e', \sigma \rangle} \quad \sigma \vdash e \Rightarrow v \quad \sigma[x \mapsto v] \vdash e' \Rightarrow v'}{\sigma \vdash (\lambda x.e')(e) \Rightarrow v'}$$

Then, we can define a smaller language FAE

Expressions	$\mathbb{E} \ni e ::= n$	(Num)
	$e + e$	(Add)
	$e \times e$	(Mul)
	$x$	(Id)
	$\lambda x.e$	(Fun)
	$e(e)$	(App)

by removing val from FVAE using the following equivalence:

$\text{val } x=e; e'$  is equivalent to  $(\lambda x.e')(e)$

## Definition (Syntactic Sugar)

Syntactic elements that can be expressed in terms of other syntactic elements are called **syntactic sugar**.

## Definition (Desugaring)

**Desugaring** is a translation for removing syntactic sugar.

$$\mathcal{D}[-] : \mathbb{E} \rightarrow \mathbb{E}$$

$\mathcal{D}[n]$	$=$	$n$	$\mathcal{D}[\text{val } x=e; e']$	$=$	$(\lambda x. \mathcal{D}[e'])(\mathcal{D}[e])$
$\mathcal{D}[e + e']$	$=$	$\mathcal{D}[e] + \mathcal{D}[e']$	$\mathcal{D}[x]$	$=$	$x$
$\mathcal{D}[e \times e']$	$=$	$\mathcal{D}[e] \times \mathcal{D}[e']$	$\mathcal{D}[\lambda x. e]$	$=$	$\lambda x. \mathcal{D}[e]$
			$\mathcal{D}[e(e')]$	$=$	$\mathcal{D}[e](\mathcal{D}[e'])$

For example,

$$\mathcal{D}[\text{val } x=42; \text{ val } y=x+1; y+2] = (\lambda x. (\lambda y. y+2)(x+1))(42)$$



We can also implement **desugaring** in Scala:

```
def desugar(expr: Expr): Expr = expr match
  case Num(n)          => Num(n)
  case Add(l, r)        => Add(desugar(l), desugar(r))
  case Mul(l, r)        => Mul(desugar(l), desugar(r))
  case Val(x, i, b)     => App(Fun(x, desugar(b)), desugar(i))
  case Id(x)            => Id(x)
  case Fun(p, b)         => Fun(p, desugar(b))
  case App(f, e)         => App(desugar(f), desugar(e))
```

Note that we need to **recursively desugar** all sub-expressions of the given expression even if they are not syntactic sugars.

Then, we can desugar the example FVAE expression as follows:

```
val e1: Expr = Expr("val x = 42; val y = x + 1; y + 2")
val e2: Expr = Expr("(x => (y => y + 2)(x + 1))(42)")
desugar(e1) == e2
```

Most programming languages have **syntactic sugar**:

- Scala

<code>for (x &lt;- list) yield x * 2</code>	$\equiv$	<code>list.map(x =&gt; x * 2)</code>
---	----------	--------------------------------------

- C++

<code>arr[i] + obj-&gt;field</code>	$\equiv$	<code>*(arr + i) + (*obj).field</code>
-------------------------------------	----------	--

- JavaScript

<code>x += y; x *= y;</code>	$\equiv$	<code>x = x + y; x = x * y;</code>
------------------------------	----------	------------------------------------

- Haskell

<code>do x &lt;- f; g x</code>	$\equiv$	<code>f &gt;&gt;= (\x -&gt; g x)</code>
--------------------------------	----------	---

- ...

## 1. Syntactic Sugar

No More `val`

F<sub>AE</sub> – Removing `val` from F<sub>VAE</sub>

Syntactic Sugar and Desugaring

## 2. Lambda Calculus

Definition

Church Encodings

Church-Turing Thesis

What is the minimal language that can express all the syntactic elements of FVAE? **Lambda calculus (LC)**!

The **lambda calculus (LC)** is a language only consisting of 1) **variables**, 2) **functions**, and 3) **applications**:

$$\begin{array}{lcl} \text{Expressions } \mathbb{E} \ni e & ::= & x \\ & & | \lambda x. e \\ & & | e(e) \end{array}$$

We already showed that the **variable definition** can be desugared to a combination of a **function definition** and an **application**:

$$\mathcal{D}[\text{val } x=e; e'] = (\lambda x. \mathcal{D}[e']) (\mathcal{D}[e])$$

Then, how can we desugar other syntactic elements of FVAE?

Let's learn the **Church encodings**!

**Church encodings** are a way to encode **data** and **operations** in the **lambda calculus (LC)**.

For example, **Church numerals** are a way to encode **natural numbers** in the **lambda calculus (LC)**.

The key idea is to encode a **natural number**  $n$  as a **function** that takes another function  $f$  and an argument  $x$  and applies  $f$  to  $x$   $n$  times:

$$\begin{aligned}\mathcal{D}[0] &= \lambda f. \lambda x. x \\ \mathcal{D}[1] &= \lambda f. \lambda x. f(x) \\ \mathcal{D}[2] &= \lambda f. \lambda x. f(f(x)) \\ \mathcal{D}[3] &= \lambda f. \lambda x. f(f(f(x))) \\ &\vdots\end{aligned}$$

$$\begin{aligned}\mathcal{D}[e_0 + e_1] &= \lambda f. \lambda x. \mathcal{D}[e_0](f)(\mathcal{D}[e_1](f)(x)) \\ \mathcal{D}[e_0 \times e_1] &= \lambda f. \lambda x. \mathcal{D}[e_0](\mathcal{D}[e_1](f))(x)\end{aligned}$$

For example,

$$\begin{aligned}\mathcal{D}[\![1 + 1]\!] &= \lambda f.\lambda x.\mathcal{D}[\![1]\!](f)(\mathcal{D}[\![1]\!](f)(x)) \\ &= \lambda f.\lambda x.(\lambda f.\lambda x.f(x))(f)((\lambda f.\lambda x.f(x))(f)(x)) \\ &= \lambda f.\lambda x.f((\lambda f.\lambda x.f(x))(f)(x)) \\ &= \lambda f.\lambda x.f(f(x)) \\ &= \mathcal{D}[\![2]\!]\end{aligned}$$

We can represent other data or operations in the **LC** using **Church encodings**, such as **integers**, **booleans**, **pairs**, **lists**, and so on.<sup>1</sup>

Let's see one more example of **Church encoding** for **booleans** and **logical operations** (i.e., **Church booleans**).

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Church\\_encoding](https://en.wikipedia.org/wiki/Church_encoding)

The key idea is to encode a **boolean**  $b$  as a **function** that takes two arguments  $t$  and  $f$  and applies  $t$  if  $b$  is true or  $f$  if  $b$  is false:

$$\begin{aligned}\mathcal{D}[\text{true}] &= \lambda t. \lambda f. t & \mathcal{D}[\text{if}(e_1) e_2 \text{ else } e_3] &= \mathcal{D}[e_1](\mathcal{D}[e_2])(\mathcal{D}[e_3]) \\ \mathcal{D}[\text{false}] &= \lambda t. \lambda f. f & \mathcal{D}[e_1 \ \&\& \ e_2] &= \mathcal{D}[e_1](\mathcal{D}[e_2])(\mathcal{D}[e_1]) \\ & & \mathcal{D}[e_1 \ || \ e_2] &= \mathcal{D}[e_1](\mathcal{D}[e_1])(\mathcal{D}[e_2]) \\ & & \mathcal{D}[\text{! } e_0] &= \lambda t. \lambda f. \mathcal{D}[e_0](f)(t)\end{aligned}$$

For example,

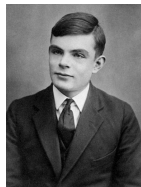
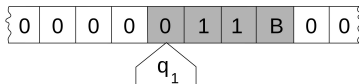
$$\begin{aligned}\mathcal{D}[\text{true} \ \&\& \ \text{false}] &= \mathcal{D}[\text{true}](\mathcal{D}[\text{false}])(\mathcal{D}[\text{true}]) \\ &= (\lambda t. \lambda f. t)(\mathcal{D}[\text{false}])(\mathcal{D}[\text{true}]) \\ &= \mathcal{D}[\text{false}]\end{aligned}$$



**Alonzo Church** invented **lambda calculus (LC)** in 1930s, and it became the foundation of **programming languages**:

$$e ::= e \mid \lambda x. e \mid e(e)$$

**Alan Turing** invented **Turing machines (TM)** in 1936, and it became the foundation of **computers**:



**Church-Turing Thesis: LC is Turing complete.**

*Any real-world computation can be translated into an equivalent computation involving a **Turing machine** or can be done using **lambda calculus**.*



## 1. Syntactic Sugar

- No More `val`

- FAE – Removing `val` from FVAE

- Syntactic Sugar and Desugaring

## 2. Lambda Calculus

- Definition

- Church Encodings

- Church-Turing Thesis

- Recursive Functions

Jihyeok Park

`jihyeok_park@korea.ac.kr`

`https://plrg.korea.ac.kr`