

Lecture 8 – Lambda Calculus

COSE212: Programming Languages

Jihyeok Park



2024 Fall

- FVAE – VAE with First-Class Functions
 - First-Class Functions
 - Concrete and Abstract Syntax
 - Interpreter and Natural Semantics with Closures
 - Static and Dynamic Scoping

- 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`

FAE – Removing `val` from FVAE

Syntactic Sugar and Desugaring

2. Lambda Calculus

Definition

Church Encodings

Church-Turing Thesis

1. Syntactic Sugar

No More `val`

FAE – 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 $[x \mapsto 1]$.

```
/* 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:

$\text{val } x = e_1; e_2$ is equivalent to $(\lambda x. e_2)(e_1)$

Why?

In general, the following two expressions are equivalent:

$\text{val } x = e_1; e_2$ is equivalent to $(\lambda x.e_2)(e_1)$

Why?

The following inference rule for the semantics of $\text{val } x = e_1; e_2$:

$$\text{VAL} \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \text{val } x = e_1; e_2 \Rightarrow v_2}$$

is equivalent to the following inference rule for the semantics of the combination $(\lambda x.e_2)(e_1)$ of an anonymous function and an application:

$$\text{App} \frac{\text{Fun} \frac{}{\sigma \vdash \lambda x.e_2 \Rightarrow \langle \lambda x.e_2, \sigma \rangle} \quad \sigma \vdash e_1 \Rightarrow v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash (\lambda x.e_2)(e_1) \Rightarrow v_2}$$

Then, we can define a smaller language FAE

Expressions $e ::= n$ (Num)
 | $e + e$ (Add)
 | $e * e$ (Mul)
 | x (Id)
 | $\lambda x.e$ (Fun)
 | $e(e)$ (App)

by removing val from FVAE using the following equivalence:

$\text{val } x = e_1; e_2$ is equivalent to $(\lambda x.e_2)(e_1)$

Definition (Syntactic Sugar)

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

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.

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

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.

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

For example, we can define the desugaring of `val` as follows:

$$\mathcal{D}[\text{val } x = e_1; e_2] = (\lambda x.e_2)(e_1)$$

Is it correct?

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.

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

For example, we can define the desugaring of `val` as follows:

$$\mathcal{D}[\text{val } x = e_1; e_2] = (\lambda x.e_2)(e_1)$$

Is it correct? **No!** Why?

$$\mathcal{D}[\text{val } x = e_1; e_2] = (\lambda x.e_2)(e_1)$$

For example,

$$\mathcal{D}[\text{val } x = 1; 2 + (\text{val } y = 3; x * y)] = \lambda x.(2 + (\text{val } y = 3; x * y))(1)$$

Without desugaring rule for addition, the expression $(\text{val } y = 3; x * y)$ in the right-hand side of the addition cannot be desugared.

$$\mathcal{D}[\text{val } x = e_1; e_2] = (\lambda x.e_2)(e_1)$$

For example,

$$\mathcal{D}[\text{val } x = 1; 2 + (\text{val } y = 3; x * y)] = \lambda x.(2 + (\text{val } y = 3; x * y))(1)$$

Without desugaring rule for addition, the expression $(\text{val } y = 3; x * y)$ in the right-hand side of the addition cannot be desugared.

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

$$\mathcal{D}[\text{val } x = e_1; e_2] = (\lambda x.\mathcal{D}[e_2])(\mathcal{D}[e_1])$$

$\mathcal{D}[n]$	$= n$	$\mathcal{D}[x]$	$= x$
$\mathcal{D}[e_1 + e_2]$	$= \mathcal{D}[e_1] + \mathcal{D}[e_2]$	$\mathcal{D}[\lambda x.e]$	$= \lambda x.\mathcal{D}[e]$
$\mathcal{D}[e_1 * e_2]$	$= \mathcal{D}[e_1] * \mathcal{D}[e_2]$	$\mathcal{D}[e_1(e_2)]$	$= \mathcal{D}[e_1](\mathcal{D}[e_2])$

$$\mathcal{D}[\text{val } x = e_1; e_2] = (\lambda x.e_2)(e_1)$$

For example,

$$\mathcal{D}[\text{val } x = 1; 2 + (\text{val } y = 3; x * y)] = \lambda x.(2 + (\text{val } y = 3; x * y))(1)$$

Without desugaring rule for addition, the expression $(\text{val } y = 3; x * y)$ in the right-hand side of the addition cannot be desugared.

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

$$\mathcal{D}[\text{val } x = e_1; e_2] = (\lambda x.\mathcal{D}[e_2])(\mathcal{D}[e_1])$$

$$\begin{array}{ll} \mathcal{D}[n] & = n & \mathcal{D}[x] & = x \\ \mathcal{D}[e_1 + e_2] & = \mathcal{D}[e_1] + \mathcal{D}[e_2] & \mathcal{D}[\lambda x.e] & = \lambda x.\mathcal{D}[e] \\ \mathcal{D}[e_1 * e_2] & = \mathcal{D}[e_1] * \mathcal{D}[e_2] & \mathcal{D}[e_1(e_2)] & = \mathcal{D}[e_1](\mathcal{D}[e_2]) \end{array}$$

Then, it can be desugared as follows:

$$\mathcal{D}[\text{val } x = 1; 2 + (\text{val } y = 3; x * y)] = \lambda x.(2 + (\lambda y.x * y)(3))(1)$$

$$\mathcal{D}[\text{val } x = e_1; e_2] = (\lambda x. \mathcal{D}[e_2])(\mathcal{D}[e_1])$$

$$\begin{array}{ll} \mathcal{D}[n] & = n & \mathcal{D}[x] & = x \\ \mathcal{D}[e_1 + e_2] & = \mathcal{D}[e_1] + \mathcal{D}[e_2] & \mathcal{D}[\lambda x. e] & = \lambda x. \mathcal{D}[e] \\ \mathcal{D}[e_1 * e_2] & = \mathcal{D}[e_1] * \mathcal{D}[e_2] & \mathcal{D}[e_1(e_2)] & = \mathcal{D}[e_1](\mathcal{D}[e_2]) \end{array}$$

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

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

$$\mathcal{D}[\text{val } x = e_1; e_2] = (\lambda x. \mathcal{D}[e_2])(\mathcal{D}[e_1])$$

$$\begin{array}{ll} \mathcal{D}[n] & = n & \mathcal{D}[x] & = x \\ \mathcal{D}[e_1 + e_2] & = \mathcal{D}[e_1] + \mathcal{D}[e_2] & \mathcal{D}[\lambda x. e] & = \lambda x. \mathcal{D}[e] \\ \mathcal{D}[e_1 * e_2] & = \mathcal{D}[e_1] * \mathcal{D}[e_2] & \mathcal{D}[e_1(e_2)] & = \mathcal{D}[e_1](\mathcal{D}[e_2]) \end{array}$$

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

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

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

```
val expr: Expr = Expr("val x = 1; 2 + (val y = 3; x * y)")
desugar(expr) == Expr("(x => 2 + (y => x * y)(3))(1)")
```

Most programming languages have **syntactic sugar**:

- Scala

```
for (x <- list) yield x * 2 ≡ list.map(x => x * 2)
```

- C++

```
arr[i] + obj->field ≡ *(arr + i) + (*obj).field
```

- JavaScript¹

```
x ||= y; x &&= y; ≡ x || (x = y); x && (x = y);
```

- Haskell

```
do x <- f; g x ≡ f >>= (\x -> g x)
```

- ...

¹<https://babeljs.io/repl>

1. Syntactic Sugar

No More `val`

F_{AE} – Removing `val` from FVAE

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?

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**:

Expressions $e ::= x$
 | $\lambda x.e$
 | $e(e)$

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**:

$$\begin{array}{l} \text{Expressions } e ::= x \\ \quad \quad \quad | \lambda x.e \\ \quad \quad \quad | e(e) \end{array}$$

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

$$\mathcal{D}[\text{val } x = e_1; e_2] = (\lambda x.\mathcal{D}[e_2])(\mathcal{D}[e_1])$$

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**:

$$\begin{array}{l} \text{Expressions } e ::= x \\ \quad \quad \quad | \lambda x.e \\ \quad \quad \quad | e(e) \end{array}$$

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

$$\mathcal{D}[\text{val } x = e_1; e_2] = (\lambda x.\mathcal{D}[e_2])(\mathcal{D}[e_1])$$

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

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**:

$$\begin{array}{l} \text{Expressions } e ::= x \\ \quad \quad \quad | \lambda x.e \\ \quad \quad \quad | e(e) \end{array}$$

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

$$\mathcal{D}[\text{val } x = e_1; e_2] = (\lambda x.\mathcal{D}[e_2])(\mathcal{D}[e_1])$$

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

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

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

Church encodings are ways 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)**.

Church encodings are ways 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:

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

$$\begin{aligned}\mathcal{D}\llbracket e_1 + e_2 \rrbracket &= \lambda f. \lambda x. \mathcal{D}\llbracket e_1 \rrbracket(f)(\mathcal{D}\llbracket e_2 \rrbracket(f)(x)) \\ \mathcal{D}\llbracket e_1 * e_2 \rrbracket &= \lambda f. \lambda x. \mathcal{D}\llbracket e_1 \rrbracket(\mathcal{D}\llbracket e_2 \rrbracket(f))(x)\end{aligned}$$

For example,

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

²https://en.wikipedia.org/wiki/Church_encoding

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.²

²https://en.wikipedia.org/wiki/Church_encoding

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.²

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

²https://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:

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

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:

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

For example,

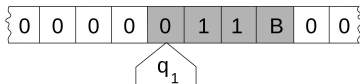
$$\begin{aligned}
 \mathcal{D}[\mathbf{true} \ \&\& \ \mathbf{false}] &= \mathcal{D}[\mathbf{true}](\mathcal{D}[\mathbf{false}])(\mathcal{D}[\mathbf{true}]) \\
 &= (\lambda t. \lambda f. t)(\mathcal{D}[\mathbf{false}])(\mathcal{D}[\mathbf{true}]) \\
 &= \mathcal{D}[\mathbf{false}]
 \end{aligned}$$



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

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

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



Church-Turing Thesis: Lambda Calculus 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**.*

<https://github.com/ku-plrg-classroom/docs/tree/main/cose212/cobalt>

- Please see above document on GitHub:
 - ① Implement `interp` function.
 - ② Implement `subExpr1` and `subExpr2` functions.
- The due date is 23:59 on Oct. 14 (Mon.).
- Please only submit `Implementation.scala` file to [Blackboard](#).

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>