Lecture 6 – First-Order Functions

COSE212: Programming Languages

Jihyeok Park

PLRG

2025 Fall

Recall

- **VAE – AE with variables**
  - Evaluation with Environments
  - Interpreter and Natural Semantics

Recall

- **VAE – AE with variables**
  - Evaluation with Environments
  - Interpreter and Natural Semantics

- In this lecture, we will learn **first-order functions**.

Recall

- **VAE – AE with variables**
  - Evaluation with Environments
  - Interpreter and Natural Semantics

- In this lecture, we will learn **first-order functions**.

- **F1VAE – VAE with first-order functions**
  - Concrete and Abstract Syntax
  - Interpreter and Natural Semantics

Contents

Contents

First-Order Functions

Let's calculate the square of several numbers in `Scala`.

```scala
3 * 3       // 9
42 * 42     // 1764
2434 * 2434 // 5924356
```

First-Order Functions

Let's calculate the square of several numbers in `Scala`.

```
3 * 3       // 9
42 * 42     // 1764
2434 * 2434 // 5924356
```

With a **first-order function**, we can avoid the repetition of the code.

```
// A `square` function that takes an integer `n` and returns its square.
def square(n: Int): Int = n * n

square(3)    // 9
square(42)   // 1764
square(2434) // 5924356
```

A **first-order function** consists of 1) a function name, 2) a list of parameters, and 3) a function body, and it can be **invoked** with arguments.

First-Order Functions

Most programming languages support **first-order functions**.

- Scala

```scala
def square(n: Int): Int = n * n
```

```scala
square(3) // 9
```

- Python

```python
def square(n): return n * n
```

```python
square(3) # 9
```

- C++

```cpp
int square(int n) { return n * n; }
```

```cpp
square(3) // 9
```

- Rust

```rust
fn square(n: i32) -> i32 { return n * n; }
```

```rust
square(3) // 9
```

- ...

Contents

F1VAE – VAE with First-Order Functions

PLRG

Now, we want to extend VAE into F1VAE with **first-order functions**.

```
/* F1VAE */
def square(n) = n * n;
square(3) + 2 // 11
```

```
/* F1VAE */
def add3(n) = n + 3;
def mul2(m) = m * 2;
mul2(add3(4)) // 14
```

F1VAE – VAE with First-Order Functions

Now, we want to extend VAE into F1VAE with **first-order functions**.

```
/* F1VAE */
def square(n) = n * n;
square(3) + 2 // 11
```

```
/* F1VAE */
def add3(n) = n + 3;
def mul2(m) = m * 2;
mul2(add3(4)) // 14
```

- An F1VAE **program** is a pair of
  1. a list of **function definitions**
  2. an **expression**

F1VAE – VAE with First-Order Functions

Now, we want to extend VAE into F1VAE with **first-order functions**.

```
/* F1VAE */
def square(n) = n * n;
square(3) + 2 // 11
```

```
/* F1VAE */
def add3(n) = n + 3;
def mul2(m) = m * 2;
mul2(add3(4)) // 14
```

- An F1VAE **program** is a pair of
  1. a list of **function definitions**
  2. an **expression**

- A **function definition** in F1VAE only has a **single parameter**.

F1VAE – VAE with First-Order Functions

Now, we want to extend VAE into F1VAE with **first-order functions**.

```
/* F1VAE */
def square(n) = n * n;
square(3) + 2 // 11
```

```
/* F1VAE */
def add3(n) = n + 3;
def mul2(m) = m * 2;
mul2(add3(4)) // 14
```

- An F1VAE **program** is a pair of
  1. a list of **function definitions**
  2. an **expression**
- A **function definition** in F1VAE only has a **single parameter**.
- We extend **expressions** with **function applications**.

Concrete Syntax

Let's define the **concrete syntax** of F1VAE in BNF:

- An F1VAE **program** is a pair of
  1. a list of **function definitions**
  2. an **expression**
- A **function definition** in F1VAE only has a **single parameter**.
- We extend **expressions** with **function applications**.

Concrete Syntax

Let's define the **concrete syntax** of F1VAE in BNF:

- An F1VAE **program** is a pair of
  1. a list of **function definitions**
  2. an **expression**
- A **function definition** in F1VAE only has a **single parameter**.
- We extend **expressions** with **function applications**.

```
// programs
<program> ::= <fdef>* <expr>
// function definitions
<fdef>    ::= "def" <id> "(" <id> ")" "=" <expr> ";"
// expressions
<expr>    ::= ...
            | "{" <expr> "}"                        // block
            | "val" <id> "=" <expr> ";" <expr>      // variable definition
            | <id>                                  // identifier lookup
            | <id> "(" <expr> ")"                   // function application
```

Abstract Syntax

Let's define the **abstract syntax** of F1VAE in BNF and its **ADT** in Scala:

$$
\begin{array}{llcll}
\text{Programs} & p & ::= & f^* \; e & (\texttt{Program}) \\
\text{Function Definitions} & \mathbb{F} \ni f & ::= & \texttt{def } x(x) \texttt{ = } e & (\texttt{FunDef}) \\
\text{Expressions} & e & ::= & \ldots & \\
 & & \mid & \texttt{val } x \texttt{ = } e; \; e & (\texttt{Val}) \\
 & & \mid & x & (\texttt{Id}) \\
 & & \mid & x(e) & (\texttt{App})
\end{array}
$$

Abstract Syntax

Let's define the **abstract syntax** of F1VAE in BNF and its **ADT** in Scala:

$$
\begin{array}{llcll}
\text{Programs} & p & ::= & f^*\ e & (\texttt{Program}) \\
\text{Function Definitions} & \mathbb{F} \ni f & ::= & \texttt{def}\ x(x)\ \texttt{=}\ e & (\texttt{FunDef}) \\
\text{Expressions} & e & ::= & \ldots & \\
 & & \mid & \texttt{val}\ x\ \texttt{=}\ e;\ e & (\texttt{Val}) \\
 & & \mid & x & (\texttt{Id}) \\
 & & \mid & x(e) & (\texttt{App})
\end{array}
$$

```scala
// programs
case class Program(fdefs: List[FunDef], expr: Expr)
// function definitions
case class FunDef(name: String, param: String, body: Expr)
// expressions
enum Expr:
  ...
  case Val(name: String, init: Expr, body: Expr)
  case Id(name: String)
  case App(fname: String, arg: Expr) // function application
```

Abstract Syntax

For example, let's **parse** the following F1VAE program:

```
/* F1VAE */
def add3(n) = n + 3;
def mul2(m) = m * 2;
mul2(add3(4))
```

Then, the following **abstract syntax tree (AST)** is produced:

```
Program(
  List(
    FunDef("add3", "n", Add(Id("n"), Num(3))),
    FunDef("mul2", "m", Mul(Id("m"), Num(2)))
  ),
  App("mul2", App("add3", Num(4)))
)
```

Contents

Let's evaluate the following F1VAE program:

```
/* F1VAE */
def add3(n) = n + 3;
def mul2(m) = m * 2;
mul2(add3(4)) // 14
```

Evaluation with Function Environments

Let's evaluate the following F1VAE program:

```
/* F1VAE */
def add3(n) = n + 3;
def mul2(m) = m * 2;
mul2(add3(4)) // 14
```

How to find the function definition of `add3` or `mul2`?

Evaluation with Function Environments

Let's evaluate the following F1VAE program:

```
/* F1VAE */
def add3(n) = n + 3;
def mul2(m) = m * 2;
mul2(add3(4)) // 14
```

How to find the function definition of `add3` or `mul2`?

We need to construct a **function environment** that maps function names to function definitions from the **list of function definitions** in a program.

$$[\texttt{add3} \mapsto f_0, \texttt{mul2} \mapsto f_1]$$

where

$$\begin{array}{lcl} f_0 & = & \texttt{def add3}(n)\ \texttt{=}\ n\ \texttt{+}\ 3 \\ f_1 & = & \texttt{def mul2}(m)\ \texttt{=}\ m\ \texttt{*}\ 2 \end{array}$$

Evaluation with Function Environments

For VAE, the interpreter takes an **expression** e with an **environment** σ and returns a number n as the result.

```
type Value = BigInt                                 // values
type Env = Map[String, Value]                       // environments
def interp(expr: Expr, env: Env): Value = ... // interpreter
```

$$\boxed{\sigma \vdash e \Rightarrow n}$$

Evaluation with Function Environments

For VAE, the interpreter takes an **expression** e with an **environment** σ and returns a number n as the result.

```
type Value = BigInt                             // values
type Env = Map[String, Value]                   // environments
def interp(expr: Expr, env: Env): Value = ... // interpreter
```

$$\boxed{\sigma \vdash e \Rightarrow n}$$

Now, we extend it to take a **function environment** Λ, a mapping from function names to function definitions, as well:

```
type Value = BigInt                             // values
type Env = Map[String, Value]                   // environments
type FEnv = Map[String, FunDef]                 // function environments
def interp(expr: Expr, env: Env, fenv: FEnv): Value = ... // interpreter
```

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

Constructing Function Environments

We have a **list of function definitions** in an F1VAE program, not a **function environment**, a mapping from function names to definitions.

```
case class Program(fdefs: List[FunDef], expr: Expr)
type FEnv = Map[String, FunDef]
```

How to construct a `FEnv` from a `List[FunDef]`?

Constructing Function Environments

We have a **list of function definitions** in an F1VAE program, not a **function environment**, a mapping from function names to definitions.

```
case class Program(fdefs: List[FunDef], expr: Expr)
type FEnv = Map[String, FunDef]
```

How to construct a `FEnv` from a `List[FunDef]`?

```
def createFEnv(fdefs: List[FunDef]): FEnv =
  fdefs.foldLeft(Map.empty)((fenv: FEnv, fdef: FunDef) => {
    val fname: String = fdef.name
    if (fenv.contains(fname)) error(s"duplicate function: $fname")
    else fenv + (fname -> fdef)
  })
```

Constructing Function Environments

We have a **list of function definitions** in an F1VAE program, not a **function environment**, a mapping from function names to definitions.

```
case class Program(fdefs: List[FunDef], expr: Expr)
type FEnv = Map[String, FunDef]
```

How to construct a `FEnv` from a `List[FunDef]`?

```
def createFEnv(fdefs: List[FunDef]): FEnv =
  fdefs.foldLeft(Map.empty)((fenv: FEnv, fdef: FunDef) => {
    val fname: String = fdef.name
    if (fenv.contains(fname)) error(s"duplicate function: $fname")
    else fenv + (fname -> fdef)
  })
```

It will throw an error if there are **duplicate function names**:

```
createFEvn(List(
  FunDef("add3", "n", Add(Id("n"), Num(3))),
  FunDef("add3", "n", Add(Num(3), Id("n"))),
)) // error: duplicate function: add3
```

Interpreter and Natural Semantics for F1VAE

For F1VAE, we need to 1) implement the **interpreter**:

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = ???
```

Interpreter and Natural Semantics for F1VAE

For F1VAE, we need to 1) implement the **interpreter**:

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = ???
```

and 2) define the **natural semantics** with environments and **function environments**:

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

Programs	p	$::=$	$f^* \; e$	(`Program`)
Function Definitions	$\mathbb{F} \ni f$	$::=$	`def` $x(x)$ `=` e	(`FunDef`)
Expressions	e	$::=$	$\ldots$	
		$\mid$	$x(e)$	(`App`)

where

Numbers	$n \in \mathbb{Z}$	(`BigInt`)
Identifiers	$x \in \mathbb{X}$	(`String`)
Environments	$\sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{Z}$	(`Env`)
Function Environments	$\Lambda \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{F}$	(`FEnv`)

Function Application

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
  ...
  case App(f, e) => ???
```

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

$$\texttt{App}\ \frac{???}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow ???}$$

Function Application

PLRG

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
  ...
  case App(f, e) =>
    val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
    ...
```

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

$$\texttt{App}\ \frac{x_0 \in \mathsf{Domain}(\Lambda) \qquad \Lambda(x_0) = \texttt{def}\ x_0(x_1)\ \texttt{=}\ e_2 \qquad \cdots}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow \texttt{???}}$$

Function Application

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
  ...
  case App(f, e) =>
    val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
    ... interp(e, env, fenv) ...
```

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

$$\texttt{App}\ \frac{x_0 \in \textsf{Domain}(\Lambda) \qquad \Lambda(x_0) = \texttt{def}\ x_0(x_1)\ \texttt{=}\ e_2 \qquad \sigma, \Lambda \vdash e_1 \Rightarrow n_1 \qquad \ldots}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow \textcolor{red}{???}}$$

Function Application

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
  ...
  case App(f, e) =>
    val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
    ... Map(fdef.param -> interp(e, env, fenv)) ...
```

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

$$\texttt{App}\ \frac{x_0 \in \mathsf{Domain}(\Lambda) \qquad \Lambda(x_0) = \texttt{def}\ x_0(x_1)\ \texttt{=}\ e_2 \qquad \sigma, \Lambda \vdash e_1 \Rightarrow n_1 \qquad \ldots [x_1 \mapsto n_1] \ldots}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow n_2}$$

Function Application

```scala
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
  ...
  case App(f, e) =>
    val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
    interp(fdef.body, Map(fdef.param -> interp(e, env, fenv)), fenv)
```

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

$$\texttt{App}\ \frac{\begin{array}{cc} x_0 \in \mathsf{Domain}(\Lambda) & \Lambda(x_0) = \texttt{def}\ x_0(x_1)\ \texttt{=}\ e_2 \\ \sigma, \Lambda \vdash e_1 \Rightarrow n_1 & [x_1 \mapsto n_1], \Lambda \vdash e_2 \Rightarrow n_2 \end{array}}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow n_2}$$

Function Application

```scala
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
  ...
  case App(f, e) =>
    val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
    interp(fdef.body, Map(fdef.param -> interp(e, env, fenv)), fenv)
```

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

$$\texttt{App}\ \frac{\begin{array}{cc} x_0 \in \mathsf{Domain}(\Lambda) & \Lambda(x_0) = \texttt{def}\ x_0(x_1)\ \texttt{=}\ e_2 \\ \sigma, \Lambda \vdash e_1 \Rightarrow n_1 & [x_1 \mapsto n_1], \Lambda \vdash e_2 \Rightarrow n_2 \end{array}}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow n_2}$$

We skip the other cases because they are only augmented with passing function environments. If you are interested, please refer to this spec:

https://github.com/ku-plrg-classroom/docs/blob/main/cose212/f1vae/f1vae-spec.pdf

Example

```
/* F1VAE */
def inc(x) = x + 1;
inc(2) // 3
```

Let's evaluate the above F1VAE program:

$$\textsc{App}\ \dfrac{\Lambda(\texttt{inc}) = f \qquad \textsc{Num}\ \dfrac{}{\varnothing, \Lambda \vdash 2 \Rightarrow 2} \qquad \textsc{Add}\ \dfrac{\textsc{Id}\ \dfrac{\texttt{x} \in \mathsf{Domain}(\sigma)}{\sigma, \Lambda \vdash \texttt{x} \Rightarrow 2} \qquad \textsc{Num}\ \dfrac{}{\sigma, \Lambda \vdash 1 \Rightarrow 1}}{\sigma, \Lambda \vdash \texttt{x + } 1 \Rightarrow 3}}{\varnothing, \Lambda \vdash \texttt{inc}(2) \Rightarrow 3}$$

where

$$\begin{array}{rcl} \Lambda & = & [\texttt{inc} \mapsto f] \\ f & = & \texttt{def inc}(\texttt{x})\texttt{ = x + }1 \\ \sigma & = & [\texttt{x} \mapsto 2] \end{array}$$

Contents

Static Scoping vs Dynamic Scoping

The current semantics is called **static scoping** (or **lexical scoping**) because a binding occurrence is determined statically without considering the function application but only the function definition.

```
/* F1VAE */
def f(x) = x + y;        // y is a free variable
{ val y = 2; f(1) } + { val y = 4; f(3) }
```

Static Scoping vs Dynamic Scoping

PLRG

The current semantics is called **static scoping** (or **lexical scoping**) because a binding occurrence is determined statically without considering the function application but only the function definition.

```
/* F1VAE */
def f(x) = x + y;        // y is a free variable
{ val y = 2; f(1) } + { val y = 4; f(3) }
```

However, we can define the semantics of F1VAE in another way by using the **dynamic scoping** instead; a binding occurrence is determined dynamically when function application is executed:

```
/* F1VAE */
def f(x) = x + y;        // y = 2 or y = 4 depending on the call-site
{ val y = 2; f(1) } + { val y = 4; f(3) } // (1 + 2) + (3 + 4) = 10
```

Static Scoping vs Dynamic Scoping

PLRG

We can design and implement the semantics of F1VAE with the **dynamic scoping** by changing the definition of the function application:

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
  ...
  case App(f, e) =>
    val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
    interp(fdef.body, env + (fdef.param -> interp(e, env, fenv)), fenv)
```

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

$$\texttt{App}\ \frac{x_0 \in \mathsf{Domain}(\Lambda) \qquad \Lambda(x_0) = \texttt{def}\ x_0(x_1)\ \texttt{=}\ e_2 \qquad \sigma, \Lambda \vdash e_1 \Rightarrow n_1 \qquad \sigma[x_1 \mapsto n_1], \Lambda \vdash e_2 \Rightarrow n_2}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow n_2}$$

Static Scoping vs Dynamic Scoping

We can design and implement the semantics of F1VAE with the **dynamic scoping** by changing the definition of the function application:

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
  ...
  case App(f, e) =>
    val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
    interp(fdef.body, env + (fdef.param -> interp(e, env, fenv)), fenv)
```

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

$$\texttt{App}\ \frac{x_0 \in \mathsf{Domain}(\Lambda) \qquad \Lambda(x_0) = \texttt{def}\ x_0(x_1) \texttt{ = } e_2 \qquad \sigma, \Lambda \vdash e_1 \Rightarrow n_1 \qquad \sigma[x_1 \mapsto n_1], \Lambda \vdash e_2 \Rightarrow n_2}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow n_2}$$

However, we will use the **static scoping** by default in this course.

Example

```
/* F1VAE */
def add(x) = x + y;
val y = 2; add(1)
```

If we evaluate the above F1VAE program with a **static scoping**:

$$\textsc{Val}\ \dfrac{\textsc{Num}\ \dfrac{}{\varnothing, \Lambda \vdash 2 \Rightarrow 2} \quad \textsc{App}\ \dfrac{\Lambda(\mathsf{add}) = f \quad \textsc{Num}\ \dfrac{}{\sigma_0, \Lambda \vdash 1 \Rightarrow 1} \quad \textsc{Add}\ \dfrac{\textsc{Id}\ \dfrac{\mathtt{x} \in \mathsf{Domain}(\sigma_1)}{\sigma_1, \Lambda \vdash \mathtt{x} \Rightarrow 1} \quad \textsc{Id}\ \dfrac{\mathtt{y} \notin \mathsf{Domain}(\sigma_1)}{\sigma_1, \Lambda \vdash \mathtt{y} \Rightarrow \textsf{FAIL}}}{\sigma_1, \Lambda \vdash \mathtt{x\ +\ y} \Rightarrow \textsf{FAIL}}}{\sigma_0, \Lambda \vdash \mathtt{add}(1) \Rightarrow \textsf{FAIL}}}{\varnothing, \Lambda \vdash \mathtt{val\ y\ =\ 2;\ add}(1) \Rightarrow \textsf{FAIL}}$$

where

$$\begin{array}{rcl} \Lambda & = & [\mathtt{add} \mapsto f] \\ f & = & \mathtt{def\ add(x)\ =\ x\ +\ y} \\ \sigma_0 & = & [\mathtt{y} \mapsto 2] \\ \sigma_1 & = & [\mathtt{x} \mapsto 1] \end{array}$$

it cannot be evaluated because `y` is a **free variable** in the `add` function.

Example

```
/* F1VAE */
def add(x) = x + y;
val y = 2; add(1)
```

However, if we evaluate it with a **dynamic scoping**:

$$\textsc{Val}\ \dfrac{\textsc{Num}\ \dfrac{}{\varnothing, \Lambda \vdash 2 \Rightarrow 2} \quad \textsc{App}\ \dfrac{\Lambda(\texttt{add}) = f \quad \textsc{Num}\ \dfrac{}{\sigma_0, \Lambda \vdash 1 \Rightarrow 1} \quad \textsc{Add}\ \dfrac{\textsc{Id}\ \dfrac{\texttt{x} \in \mathsf{Domain}(\sigma_1)}{\sigma_1, \Lambda \vdash \texttt{x} \Rightarrow 1} \quad \textsc{Id}\ \dfrac{\texttt{y} \in \mathsf{Domain}(\sigma_1)}{\sigma_1, \Lambda \vdash \texttt{y} \Rightarrow 2}}{\sigma_1, \Lambda \vdash \texttt{x + y} \Rightarrow 3}}{\sigma_0, \Lambda \vdash \texttt{add}(1) \Rightarrow 3}}{\varnothing, \Lambda \vdash \texttt{val y = } 2\texttt{; add}(1) \Rightarrow 3}$$

where

$$\begin{array}{rcl} \Lambda & = & [\texttt{add} \mapsto f] \\ f & = & \texttt{def add}(\texttt{x}) \texttt{ = x + y} \\ \sigma_0 & = & [\texttt{y} \mapsto 2] \\ \sigma_1 & = & \sigma_0[\texttt{x} \mapsto 1] = [\texttt{y} \mapsto 2, \texttt{x} \mapsto 1,] \end{array}$$

it evaluates to 3 because y is **dynamically bound** to 2.

Summary

$$
\begin{array}{llcll}
\text{Programs} & p & ::= & f^* \; e & (\texttt{Program}) \\
\text{Function Definitions} & \mathbb{F} \ni f & ::= & \texttt{def} \; x(x) \; \texttt{=} \; e & (\texttt{FunDef}) \\
\text{Expressions} & e & ::= & \ldots & \\
 & & | & x(e) & (\texttt{App})
\end{array}
$$

```scala
type FEnv = Map[String, FunDef]
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
  ...
  case App(f, e) =>
    val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
    interp(fdef.body, Map(fdef.param -> interp(e, env, fenv)), fenv)
```

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

$$
\texttt{App}\;\frac{\begin{array}{cc} x_0 \in \mathsf{Domain}(\Lambda) & \Lambda(x_0) = \texttt{def}\; x_0(x_1) \;\texttt{=}\; e_2 \\ \sigma, \Lambda \vdash e_1 \Rightarrow n_1 & [x_1 \mapsto n_1], \Lambda \vdash e_2 \Rightarrow n_2 \end{array}}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow n_2}
$$

Exercise #3

https://github.com/ku-plrg-classroom/docs/tree/main/cose212/f1vae

- Please see above document on GitHub:
  - Implement `interp` function.
  - Implement `interpDS` function.

- It is just an exercise, and you **don't need to submit** anything.
- However, some exam questions might be related to this exercise.

Next Lecture

- First-Class Functions

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr