Lecture 13 – Lazy Evaluation

COSE212: Programming Languages

Jihyeok Park

PLRG

2024 Fall

Recall

- We learned two different evaluation strategies, **call-by-value** and **call-by-reference**, in the previous lecture.
    - **Call-by-value** (CBV) eagerly evaluates the arguments and passes the evaluated values to the function.
    - **Call-by-reference** (CBR) passes the references (i.e., addresses) of the arguments to the function.

Recall

- We learned two different evaluation strategies, **call-by-value** and **call-by-reference**, in the previous lecture.
    - **Call-by-value** (CBV) eagerly evaluates the arguments and passes the evaluated values to the function.
    - **Call-by-reference** (CBR) passes the references (i.e., addresses) of the arguments to the function.

- In this lecture, we will learn another evaluation strategy called **lazy evaluation**, while the previous two are called **eager evaluation**.
    - **Call-by-name** (CBN)
    - **Call-by-need** (CBN')

- **LFAE** – FAE with Lazy Evaluation
    - Interpreter and Natural Semantics

Contents

Contents

Lazy Evaluation

So far, all the languages we have defined are based on the **eager evaluation** strategy; all the expressions are eagerly evaluated regardless of whether they are really needed or not.

Lazy Evaluation

So far, all the languages we have defined are based on the **eager evaluation** strategy; all the expressions are eagerly evaluated regardless of whether they are really needed or not.

Consider following two expressions in FAE:

```
/* FAE */
val x = 1 + 2;
val y = z + 3;                    // error -- free identifier: z
x * 2
```

Lazy Evaluation

So far, all the languages we have defined are based on the **eager evaluation** strategy; all the expressions are eagerly evaluated regardless of whether they are really needed or not.

Consider following two expressions in FAE:

```
/* FAE */
val x = 1 + 2;
val y = z + 3;                    // error -- free identifier: z
x * 2
```

```
/* FAE */
val f = x => y => x * 2;
f(1 + 2)(1 * 2 * ... * 10000000)  // too slow -- unnecessary computation
```

Lazy Evaluation

So far, all the languages we have defined are based on the **eager evaluation** strategy; all the expressions are eagerly evaluated regardless of whether they are really needed or not.

Consider following two expressions in FAE:

```
/* FAE */
val x = 1 + 2;
val y = z + 3;                    // error -- free identifier: z
x * 2
```

```
/* FAE */
val f = x => y => x * 2;
f(1 + 2)(1 * 2 * ... * 10000000)  // too slow -- unnecessary computation
```

If we can **delay** the evaluation of expressions until their results are **used**, we can **avoid** unnecessary computations and errors.

Lazy Evaluation

So far, all the languages we have defined are based on the **eager evaluation** strategy; all the expressions are eagerly evaluated regardless of whether they are really needed or not.

Consider following two expressions in FAE:

```
/* FAE */
val x = 1 + 2;
val y = z + 3;                    // error -- free identifier: z
x * 2
```

```
/* FAE */
val f = x => y => x * 2;
f(1 + 2)(1 * 2 * ... * 10000000)  // too slow -- unnecessary computation
```

If we can **delay** the evaluation of expressions until their results are **used**, we can **avoid** unnecessary computations and errors.

This is called **lazy evaluation**!

Lazy Evaluation

For example, Scala supports **lazy evaluation** for 1) **immutable variables** (`val`) with the `lazy` keyword

```
val x = 1 + 2;
lazy val y = 5 / 0          // delay its evaluation until `y` is used
x * 2                       // 6
```

Lazy Evaluation

For example, Scala supports **lazy evaluation** for 1) **immutable variables** (val) with the lazy keyword

```
val x = 1 + 2;
lazy val y = 5 / 0        // delay its evaluation until `y` is used
x * 2                     // 6
```

and 2) **parameters** with the prefix =>.

```
// delay the evaluation of the second argument until `y` is used
def f(x: Int, y: => Int): Int = x * 2

f(1 + 2, { Thread.sleep(5000); 42 })  // 6
```

Lazy Evaluation

For example, Scala supports **lazy evaluation** for 1) **immutable variables** (`val`) with the `lazy` keyword

```
val x = 1 + 2;
lazy val y = 5 / 0        // delay its evaluation until `y` is used
x * 2                     // 6
```

and 2) **parameters** with the prefix =>.

```
// delay the evaluation of the second argument until `y` is used
def f(x: Int, y: => Int): Int = x * 2

f(1 + 2, { Thread.sleep(5000); 42 })  // 6
```

The expression `5 / 0` throwing a division by zero error is not evaluated because the variable `y` is **not used**.

Lazy Evaluation

For example, Scala supports **lazy evaluation** for 1) **immutable variables** (`val`) with the `lazy` keyword

```
val x = 1 + 2;
lazy val y = 5 / 0         // delay its evaluation until `y` is used
x * 2                      // 6
```

and 2) **parameters** with the prefix =>.

```
// delay the evaluation of the second argument until `y` is used
def f(x: Int, y: => Int): Int = x * 2

f(1 + 2, { Thread.sleep(5000); 42 })  // 6
```

The expression `5 / 0` throwing a division by zero error is not evaluated because the variable `y` is **not used**.

The expression `{ Thread.sleep(5000); 42 }` taking 5 seconds to evaluate is not evaluated because the parameter `y` is **not used**.

Lazy Evaluation

Many programming languages support **lazy evaluation** for many reasons.

- **Short-circuit Evaluation**: It could avoid unnecessary computations for boolean expressions.

```
true  && ((5 / 0) < 1)        // error  -- division by zero
false && ((5 / 0) < 1)        // false  -- (5/0)<1 is not evaluated
true  || ((5 / 0) < 1)        // true   -- (5/0)<1 is not evaluated
false || ((5 / 0) < 1)        // error  -- division by zero
```

Lazy Evaluation

Many programming languages support **lazy evaluation** for many reasons.

- **Short-circuit Evaluation**: It could avoid unnecessary computations for boolean expressions.

```
true  && ((5 / 0) < 1)        // error  -- division by zero
false && ((5 / 0) < 1)        // false  -- (5/0)<1 is not evaluated
true  || ((5 / 0) < 1)        // true   -- (5/0)<1 is not evaluated
false || ((5 / 0) < 1)        // error  -- division by zero
```

(Note that the operators `&` and `|` are similar to `&&` and `||` but do not support short-circuit evaluation in Scala.)

```
true  & ((5 / 0) < 1)         // error  -- division by zero
false & ((5 / 0) < 1)         // error  -- division by zero
true  | ((5 / 0) < 1)         // error  -- division by zero
false | ((5 / 0) < 1)         // error  -- division by zero
```

Lazy Evaluation

Many programming languages support **lazy evaluation** for many reasons.

- **Short-circuit Evaluation**: It could avoid unnecessary computations for boolean expressions.

```
true  && ((5 / 0) < 1)        // error  -- division by zero
false && ((5 / 0) < 1)        // false  -- (5/0)<1 is not evaluated
true  || ((5 / 0) < 1)        // true   -- (5/0)<1 is not evaluated
false || ((5 / 0) < 1)        // error  -- division by zero
```

(Note that the operators & and | are similar to && and || but do not support short-circuit evaluation in Scala.)

```
true  & ((5 / 0) < 1)         // error  -- division by zero
false & ((5 / 0) < 1)         // error  -- division by zero
true  | ((5 / 0) < 1)         // error  -- division by zero
false | ((5 / 0) < 1)         // error  -- division by zero
```

Most programming languages (e.g., C++, Java, Python, JavaScript, and Scala) support **short-circuit evaluation** for boolean expressions.

Lazy Evaluation

- **Optimization**: It could optimize the performance by avoiding unnecessary computations.

```
def f(x: Int, y: => Int): Int = if (x < 0) 0 else x * y
f(42, { Thread.sleep(5000); 42 }) // second arg. is evaluated
f(-7, { Thread.sleep(5000); 42 }) // second arg. is NOT evaluated
```

Lazy Evaluation

- **Optimization**: It could optimize the performance by avoiding unnecessary computations.

```
def f(x: Int, y: => Int): Int = if (x < 0) 0 else x * y
f(42, { Thread.sleep(5000); 42 }) // second arg. is evaluated
f(-7, { Thread.sleep(5000); 42 }) // second arg. is NOT evaluated
```

In fact, we already utilized lazy evaluation in our interpreter:

```
// The definition of `getOrElse` method in `Map`
def getOrElse[V1 >: V](key: K, default: => V1): V1 = ...

// The implementation of interpreter
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Id(x) => env.getOrElse(x, error(s"free identifier: $x"))
```

The second argument `error(...)` is not evaluated when `env` does not have the key for the string stored in `x`.

Lazy Evaluation

- **Infinite Data Structures**: It makes it possible to define and manipulate infinite data structures.

```
val nats: LazyList[BigInt] = 0 #:: nats.map(_ + 1)
// nats = 0 #:: (... - [not yet eval])
```

Lazy Evaluation

- **Infinite Data Structures**: It makes it possible to define and manipulate infinite data structures.

```
val nats: LazyList[BigInt] = 0 #:: nats.map(_ + 1)
// nats = 0 #:: (... - [not yet eval])
nats(3) // 3
// nats = 0 #:: 1 #:: 2 #:: 3 #:: (... - [not yet eval])
```

Lazy Evaluation

- **Infinite Data Structures**: It makes it possible to define and manipulate infinite data structures.

```
val nats: LazyList[BigInt] = 0 #:: nats.map(_ + 1)
// nats = 0 #:: (... - [not yet eval])
nats(3) // 3
// nats = 0 #:: 1 #:: 2 #:: 3 #:: (... - [not yet eval])
nats(1) // 1
// nats = 0 #:: 1 #:: 2 #:: 3 #:: (... - [not yet eval])
```

Lazy Evaluation

- **Infinite Data Structures**: It makes it possible to define and manipulate infinite data structures.

```
val nats: LazyList[BigInt] = 0 #:: nats.map(_ + 1)
// nats = 0 #:: (... - [not yet eval])
nats(3) // 3
// nats = 0 #:: 1 #:: 2 #:: 3 #:: (... - [not yet eval])
nats(1) // 1
// nats = 0 #:: 1 #:: 2 #:: 3 #:: (... - [not yet eval])
nats(4) // 4
// nats = 0 #:: 1 #:: 2 #:: 3 #:: 4 #:: (... - [not yet eval])
```

Lazy Evaluation

- **Infinite Data Structures**: It makes it possible to define and manipulate infinite data structures.

```
val nats: LazyList[BigInt] = 0 #:: nats.map(_ + 1)
// nats = 0 #:: (... - [not yet eval])
nats(3) // 3
// nats = 0 #:: 1 #:: 2 #:: 3 #:: (... - [not yet eval])
nats(1) // 1
// nats = 0 #:: 1 #:: 2 #:: 3 #:: (... - [not yet eval])
nats(4) // 4
// nats = 0 #:: 1 #:: 2 #:: 3 #:: 4 #:: (... - [not yet eval])
```

It is useful for **dynamic programming** (e.g., memoization) and **stream processing** (e.g., infinite data streams).

Lazy Evaluation

- **Infinite Data Structures**: It makes it possible to define and manipulate infinite data structures.

```
val nats: LazyList[BigInt] = 0 #:: nats.map(_ + 1)
// nats = 0 #:: (... - [not yet eval])
nats(3) // 3
// nats = 0 #:: 1 #:: 2 #:: 3 #:: (... - [not yet eval])
nats(1) // 1
// nats = 0 #:: 1 #:: 2 #:: 3 #:: (... - [not yet eval])
nats(4) // 4
// nats = 0 #:: 1 #:: 2 #:: 3 #:: 4 #:: (... - [not yet eval])
```

It is useful for **dynamic programming** (e.g., memoization) and **stream processing** (e.g., infinite data streams).

Many functional languages (e.g., Haskell) support it.

```
let nats = 0 : map (+1) nats
take 10 nats              -- [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Contents

LFAE – FAE with Lazy Evaluation

Now, let's extend FAE into LFAE to support **lazy evaluation**. (Assume that `val` is supported in FAE as syntactic sugar.)

```
/* LFAE */
val x = 1 + 2;
val y = z + 3;
x * 2
// error (FAE) vs. 6 (LFAE)
```

```
/* LFAE */
val f = x => y => x * 2;
f(1 + 2)(z + 3)
// error (FAE) vs. 6 (LFAE)
```

LFAE – FAE with Lazy Evaluation

Now, let's extend FAE into LFAE to support **lazy evaluation**. (Assume that `val` is supported in FAE as syntactic sugar.)

```
/* LFAE */
val x = 1 + 2;
val y = z + 3;
x * 2
// error (FAE) vs. 6 (LFAE)
```

```
/* LFAE */
val f = x => y => x * 2;
f(1 + 2)(z + 3)
// error (FAE) vs. 6 (LFAE)
```

There is no change in the syntax but we need to revise the semantics to support **lazy evaluation** rather than **eager evaluation**.

LFAE – FAE with Lazy Evaluation

Now, let's extend FAE into LFAE to support **lazy evaluation**. (Assume that `val` is supported in FAE as syntactic sugar.)

```
/* LFAE */
val x = 1 + 2;
val y = z + 3;
x * 2
// error (FAE) vs. 6 (LFAE)
```

```
/* LFAE */
val f = x => y => x * 2;
f(1 + 2)(z + 3)
// error (FAE) vs. 6 (LFAE)
```

There is no change in the syntax but we need to revise the semantics to support **lazy evaluation** rather than **eager evaluation**.

In LFAE, we want to **delay** the evaluation of **argument expressions** until their values are really **needed** for the computation.

LFAE – FAE with Lazy Evaluation

Now, let's extend FAE into LFAE to support **lazy evaluation**. (Assume that `val` is supported in FAE as syntactic sugar.)

```
/* LFAE */
val x = 1 + 2;
val y = z + 3;
x * 2
// error (FAE) vs. 6 (LFAE)
```

```
/* LFAE */
val f = x => y => x * 2;
f(1 + 2)(z + 3)
// error (FAE) vs. 6 (LFAE)
```

There is no change in the syntax but we need to revise the semantics to support **lazy evaluation** rather than **eager evaluation**.

In LFAE, we want to **delay** the evaluation of **argument expressions** until their values are really **needed** for the computation.

Note that the **immutable variables** (`val`) are supported as syntactic sugar of combination of function definitions and applications.

Interpreter and Natural Semantics

For LFAE, we need to 1) implement the **interpreter** with environments:

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

Interpreter and Natural Semantics

For LFAE, we need to 1) implement the **interpreter** with environments:

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

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

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

with a new kind of values called **expression values** for lazy evaluation.

$$\begin{array}{rrll}
\text{Values} & \mathbb{V} \ni v ::= & n & (\texttt{NumV}) \\
& | & \langle \lambda x.e, \sigma \rangle & (\texttt{CloV}) \\
& | & \langle\!\langle e, \sigma \rangle\!\rangle & (\texttt{ExprV})
\end{array}$$

```scala
enum Value:
  case NumV(n: BigInt)
  case CloV(p: String, b: Expr, e: Env)
  case ExprV(e: Expr, env: Env)  // for lazy evaluation
```

Interpreter and Natural Semantics

We need to keep not only expressions but also environments in the **expression values** for correct evaluation.

Interpreter and Natural Semantics

We need to keep not only expressions but also environments in the **expression values** for correct evaluation. For example,

```
/* LFAE */
val y = 3;
// [ y -> 3 ]
val inc = x => {
  // [ x -> y * 2, y -> 3 ]
  x + 1     // (3 * 2) + 1 = 7 bug expected (5 * 2) + 1 = 11
};
val y = 5;
// [ y -> 5 ]
inc(y * 2)
```

If we pass only the argument expression `y * 2`, `y` is evaluated to 3 in the body of `inc` rather than 5.

Interpreter and Natural Semantics

We need to keep not only expressions but also environments in the **expression values** for correct evaluation. For example,

```
/* LFAE */
val y = 3;
// [ y -> 3 ]
val inc = x => {
  // [ x -> y * 2, y -> 3 ]
  x + 1     // (3 * 2) + 1 = 7 bug expected (5 * 2) + 1 = 11
};
val y = 5;
// [ y -> 5 ]
inc(y * 2)
```

If we pass only the argument expression `y * 2`, `y` is evaluated to 3 in the body of `inc` rather than 5.

It means that we need to capture the current environment in the expression value similar to the closure value.

Function Application

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> interp(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma'[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

Function Application

```scala
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> interp(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma'[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

We want to **delay** the evaluation of the **argument expression** e_1 until the **parameter** x is used in the body expression e_2.

Function Application

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> interp(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma'[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

We want to **delay** the evaluation of the **argument expression** e_1 until the **parameter** x is used in the body expression e_2.

Let's define an **expression value** $\langle\langle e_1, \sigma \rangle\rangle$ to delay the evaluation of the argument expression e_1.

Function Application

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> ExprV(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma'[x \mapsto \langle\!\langle e_1, \sigma \rangle\!\rangle] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

Function Application

```scala
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> ExprV(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma'[x \mapsto \langle\!\langle e_1, \sigma \rangle\!\rangle] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

Then, when they are actually **evaluated**?

Function Application

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> ExprV(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma'[x \mapsto \langle\!\langle e_1, \sigma \rangle\!\rangle] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

Then, when they are actually **evaluated**? It depends on our design choice!

Function Application

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> ExprV(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma'[x \mapsto \langle\!\langle e_1, \sigma \rangle\!\rangle] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

Then, when they are actually **evaluated**? It depends on our design choice!

In LFAE, we will evaluate the argument expression e_1 when their values are really **needed for the computation**.

Addition

```
type BOp[T] = (T, T) => T
def numBOp(op: BOp[BigInt], x: String): BOp[Value] = (l, r) =>
  (l, r) match
    case (NumV(l), NumV(r)) => NumV(op(l, r))
    case (l, r) => error(s"invalid operation: ${l.str} $x ${r.str}")
val numAdd: BOp[Value] = numBOp(_ + _, "+")

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Add(l, r) => numAdd(interp(l, env), interp(r, env))
```

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

$$\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2}$$

Is it okay?

Addition

```
type BOp[T] = (T, T) => T
def numBOp(op: BOp[BigInt], x: String): BOp[Value] = (l, r) =>
  (l, r) match
    case (NumV(l), NumV(r)) => NumV(op(l, r))
    case (l, r) => error(s"invalid operation: ${l.str} $x ${r.str}")
val numAdd: BOp[Value] = numBOp(_ + _, "+")

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Add(l, r) => numAdd(interp(l, env), interp(r, env))
```

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

$$\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1\ \texttt{+}\ e_2 \Rightarrow n_1 + n_2}$$

Is it okay? No!

Addition

```
type BOp[T] = (T, T) => T
def numBOp(op: BOp[BigInt], x: String): BOp[Value] = (l, r) =>
  (l, r) match
    case (NumV(l), NumV(r)) => NumV(op(l, r))
    case (l, r) => error(s"invalid operation: ${l.str} $x ${r.str}")
val numAdd: BOp[Value] = numBOp(_ + _, "+")

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Add(l, r) => numAdd(interp(l, env), interp(r, env))
```

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

$$\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2}$$

Is it okay? No! If evaluation results of e_1 or e_2 are expression values, we need to evaluate them to get actual values for addition.

Addition

```
type BOp[T] = (T, T) => T
def numBOp(op: BOp[BigInt], x: String): BOp[Value] = (l, r) =>
  (l, r) match
    case (NumV(l), NumV(r)) => NumV(op(l, r))
    case (l, r) => error(s"invalid operation: ${l.str} $x ${r.str}")
val numAdd: BOp[Value] = numBOp(_ + _, "+")

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Add(l, r) => numAdd(interp(l, env), interp(r, env))
```

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

$$\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2}$$

Is it okay? No! If evaluation results of e_1 or e_2 are expression values, we need to evaluate them to get actual values for addition.

Let's define the **strict evaluation** for values to get its actual value.

Strict Evaluation for Values

The **strict evaluation** for values

1. evaluates the expression value to get its actual value, or
2. returns the value itself.

$$\boxed{v \Downarrow v}$$

$$\texttt{StrictExpr}\ \frac{\sigma \vdash e \Rightarrow v \qquad v \Downarrow v'}{\langle\!\langle e, \sigma \rangle\!\rangle \Downarrow v'}$$

$$\texttt{StrictNum}\ \frac{}{n \Downarrow n} \qquad \texttt{StrictClo}\ \frac{}{\langle \lambda x.e, \sigma \rangle \Downarrow \langle \lambda x.e, \sigma \rangle}$$

Since the evaluation of the expression value $\langle\!\langle e, \sigma \rangle\!\rangle$ may be an expression value as well. We need to recursively evaluate the expression value until we get the actual value (a number or a closure).

```scala
def strict(v: Value): Value = v match
  case ExprV(e, env) => strict(interp(e, env))
  case _             => v
```

Addition

```
type BOp[T] = (T, T) => T
def numBOp(op: BOp[BigInt], x: String): BOp[Value] = (l, r) =>
  (l, r) match
    case (NumV(l), NumV(r)) => NumV(op(l, r))
    case (l, r) => error(s"invalid operation: ${l.str} $x ${r.str}")
val numAdd: BOp[Value] = numBOp(_ + _, "+")

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Add(l, r) => numAdd(interp(l, env), interp(r, env))
```

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

$$\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2}$$

Now let's apply the **strict evaluation** for values to get the actual values of operands e_1 and e_2 for addition.

Addition

```
type BOp = (BigInt, BigInt) => BigInt
def numBOp(x: String)(op: BOp)(l: Value, r: Value): Value =
  (strict(l), strict(r)) match
    case (NumV(l), NumV(r)) => NumV(op(l, r))
    case (l, r) => error(s"invalid operation: ${l.str} $x ${r.str}")
val numAdd: BOp[Value] = numBOp(_ + _, "+")

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Add(l, r) => numAdd(interp(l, env), interp(r, env))
```

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

$$\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad v_2 \Downarrow n_2}{\sigma \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2}$$

Multiplication

```
type BOp = (BigInt, BigInt) => BigInt
def numBOp(x: String)(op: BOp)(l: Value, r: Value): Value =
  (strict(l), strict(r)) match
    case (NumV(l), NumV(r)) => NumV(op(l, r))
    case (l, r) => error(s"invalid operation: ${l.str} $x ${r.str}")
val numMul: BOp[Value] = numBOp(_ * _, "*")

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Mul(l, r) => numMul(interp(l, env), interp(r, env))
```

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

$$\texttt{Mul}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad v_2 \Downarrow n_2}{\sigma \vdash e_1 \texttt{ * } e_2 \Rightarrow n_1 \times n_2}$$

Similarly, we need to perform strict evaluation for both operands for multiplication as well.

Identifier Lookup

```
def interp(expr: Expr, env: Env): Value = expr match
  case Id(x) => env.getOrElse(x, error(s"free identifier: $x"))
```

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

$$\texttt{Id}\ \frac{x \in \mathsf{Domain}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)}$$

We will not perform strict evaluation for the value of identifier lookup because we can just pass the value without knowing its actual value.

Function Application (Cont.)

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> ExprV(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma'[x \mapsto \langle\!\langle e_1, \sigma \rangle\!\rangle] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

Function Application (Cont.)

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> ExprV(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma'[x \mapsto \langle\!\langle e_1, \sigma \rangle\!\rangle] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

In the following example, the variable `f` has an expression value $\langle\!\langle \lambda x.(x + 1), \varnothing \rangle\!\rangle$ rather than a closure value.

```
/* LFAE */
(f => f(1))(x => x+1) // error -- not a function (expression value in f)
```

Function Application (Cont.)

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => strict(interp(f, env)) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> ExprV(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow v_0 \qquad v_0 \Downarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma'[x \mapsto \langle\!\langle e_1, \sigma \rangle\!\rangle] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

```
/* LFAE */
(f => f(1))(x => x+1) // 2
```

It means that we need to perform the **strict evaluation** for the value of function expression to get the actual value.

Contents

Call-by-Name (CBN) vs. Call-by-Need (CBN')

There are two different evaluation strategies for **lazy evaluation**.

Call-by-Name (CBN) vs. Call-by-Need (CBN')

There are two different evaluation strategies for **lazy evaluation**.

Call-by-Name (CBN) evaluation strategy evaluates delayed expressions **multiple times** if they are used multiple times (e.g., parameters defined with the prefix =>)

Call-by-Name (CBN) vs. Call-by-Need (CBN')

There are two different evaluation strategies for **lazy evaluation**.

Call-by-Name (CBN) evaluation strategy evaluates delayed expressions **multiple times** if they are used multiple times (e.g., parameters defined with the prefix =>)

```scala
def inc(x: Int): Int = { println("inc"); x + 1 }
def mul5(x: => Int): Int = x + x + x + x + x
mul5(inc(1))          // 10 and prints "inc" 5 times
```

Call-by-Name (CBN) vs. Call-by-Need (CBN')

PLRG

There are two different evaluation strategies for **lazy evaluation**.

Call-by-Name (CBN) evaluation strategy evaluates delayed expressions **multiple times** if they are used multiple times (e.g., parameters defined with the prefix =>)

```
def inc(x: Int): Int = { println("inc"); x + 1 }
def mul5(x: => Int): Int = x + x + x + x + x
mul5(inc(1))        // 10 and prints "inc" 5 times
```

Call-by-Need (CBN') evaluation strategy is a **memoized** version of CBN, which evaluates delayed expressions only **once** at the first time they are used and then **reuses** the result (e.g., immutable variables (`val`) defined with `lazy` keyword).

```
def inc(x: Int): Int = { println("inc"); x + 1 }
lazy val x: Int = inc(1)
x + x + x + x + x   // 10 and prints "inc" only once
```

Call-by-Name (CBN) vs. Call-by-Need (CBN')

In purely functional languages, CBN' is **equivalent** to CBN and only has **performance benefits** because it avoids unnecessary re-evaluations.

Call-by-Name (CBN) vs. Call-by-Need (CBN')

In purely functional languages, CBN' is **equivalent** to CBN and only has **performance benefits** because it avoids unnecessary re-evaluations.

However, with **mutation**, CBN' is **not equivalent** to CBN because it evaluates function arguments **only once** the first time they are used, and thus, it may lead to **different** results:

```scala
var count: Int = 0
def get: Int = { count += 1; count }
def f(x: => Int): Int = { x + x + x }   // Call-by-Name
f(get)                                  // 1 + 2 + 3 = 6
```

```scala
var count: Int = 0
def get: Int = { count += 1; count }
lazy val x: Int = get                   // Call-by-Need
x + x + x                               // 1 + 1 + 1 = 3
```

Interpreter for Call-by-Need (CBN')

While the original LFAE uses **call-by-name** evaluation strategy, we can easily modify it to use **call-by-need** evaluation strategy as follows:

```
enum Value:
  ...
  case ExprV(e: Expr, env: Env, var value: Option[Value]) // For caching
def strict(v: Value): Value = v match
  case ev @ ExprV(e, env, v) => v match
    case Some(cache) => cache        // Reuse cached value
    case None =>                     // The first use
      val cache = interp(e, env)     // Evaluate the expression
      ev.value = Some(cache)         // Cache the value
      cache                          // Return the value
  case _ => v
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => strict(interp(f, env)) match
    // Initialize `value` with `None` to represent no caching
    case CloV(p,b,fenv) => interp(b, fenv + (p -> ExprV(e, env, None)))
    case v              => error(s"not a function: ${v.str}")
```

Exercise #8

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

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

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

Midterm Exam

- The midterm exam will be given in class.
- **Date:** 18:30 – 21:00 (150 min.), October 23 (Wed.).
- **Location:** 205, Woojung Hall of Informatics (우정정보관)
- **Coverage:** Lectures 1 – 13
- **Format:** closed book and closed notes
  - Define the syntax or semantics of extended language features.
  - Write the evaluation results of given expressions.
  - Yes/No questions about concepts in programming languages.
  - Fill-in-the-blank questions about the PL concepts.
  - etc.
- Note that there is **no class** on **October 21 (Mon.)**.

Summary

1. Lazy Evaluation

2. LFAE – FAE with Lazy Evaluation
    - Interpreter and Natural Semantics
    - Function Application
    - Strict Evaluation for Values
    - Addition and Multiplication
    - Identifier Lookup
    - Function Application (Cont.)

3. Call-by-Name (CBN) vs. Call-by-Need (CBN')
    - Interpreter for Call-by-Need (CBN')

Next Lecture

- Continuations

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