



Programming Languages

Departamento de Engenharia Informática, Técnico Lisboa

MEIC P4 24.25 (v5May2025)

M2

© Luís Caires (luís.aires@tecnico.ulisboa.pt | <https://luiscaires.org>)

The Programming Language L0

```
let x=1 ;
let f = fn y => {
    let k = x*2;
    y+x*k
};
let g = fn x, u => { u(x) + f(x) };
g ( f(3), f )
;;
```

```
public class L0int {

    public static void main(String args[]) {
        Parser parser = new Parser(System.in);
        ASTNode exp;

        System.out.println("L0 interpreter PL MEIC 2024/25 (v0.0)\n");

        while (true) {
            try {
                System.out.print("# ");
                exp = parser.Start();
                if (exp==null) System.exit(0);
                IValue v = exp.eval(new Environment<IValue>());
                System.out.println(v.toStr());
            } catch (ParseException e) {
                System.out.println("Syntax Error.");
                parser.ReInit(System.in);
            } catch (Exception e) {
                e.printStackTrace();
                parser.ReInit(System.in);
            }
        }
    }
}
```

In this module we will rigorously define the operational semantics of L0, and discuss its interpreter implementation.

Plan of the course: Growing a Language

- L0 - base values (int, bool, ... and operations), declarations, and high-order functions
- L1 = L0 + state
- L2 = L1 + simple types
- L3 = L2 + product types + sum types + recursive types + type declarations
- L4 = L3 + classes and objects
- L5 = L4 + affine and linear types

Along the way we will specify the language's semantics formally, cover proof techniques of some their properties, while growing an interpreter step by step.

M2 Themes

- Naming
- Binding, Bound and Free identifiers, Scope
- Safe Substitution
- Reduction Semantics
- Big Step Semantics
- Environment / Closure Semantics
- Implementation of Environments

The Programming Language L0

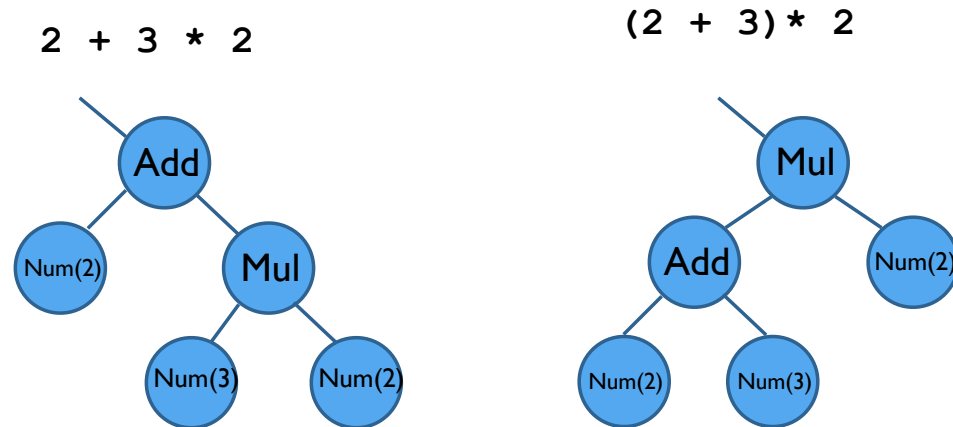
- Abstract Syntax (given by a **constructor signature**)

```
num:    int  $\rightarrow$  L0
bool:   boolean  $\rightarrow$  L0
id:     string  $\rightarrow$  L0
add:    L0  $\times$  L0  $\rightarrow$  L0
mul:    L0  $\times$  L0  $\rightarrow$  L0
....
if:     L0  $\times$  L0  $\rightarrow$  L0
let:    list[string  $\times$  L0]  $\times$  L0  $\rightarrow$  L0
fun:    string  $\times$  L0  $\rightarrow$  L0
app:    L0  $\times$  L0  $\rightarrow$  L0
```

Abstract Syntax

Abstract syntax describes the structured core structure and elements of programs and expressions, while **concrete syntax** defines the specific, visible representation (like it shows in actual textual code) of that structure.

An abstract syntax tree (AST) represents the structure of expressions / programs in terms composition of **functional abstract constructors**, as a tree-like structure.



Abstract Syntax vs. Concrete Syntax (examples)

- Numeral literals
 - decimal notation : 12
 - hexadecimal notation : 0x0C
 - **abstract syntax:** num(12)
- Identifiers
 - C: xpto
 - bash: %A
 - **abstract syntax:** id("A")
- Assignment
 - C: x = 2
 - OCAML: x := 2
 - **abstract syntax:** assign(id("x"),num(2))

Abstract Syntax vs. Concrete Syntax (examples)

- Algebraic Expressions
 - common: $2*3+2$
 - RPN: $2\ 3\ *\ 2\ +$
 - Lisp: $(+ (*\ 2\ 3)\ 2)$
 - **abstract syntax:** `add(mul(num(2), num(3)), num(2))`
- Block
 - C: `{ S1 S2 ... Sn }`
 - Python: `\tab S1 \tab S2 ... \tab Sn`
 - **abstract syntax:** `block(S1,S2,...,Sn)` or `seq(S1, seq(S2, seq (...)))`
- While loop:
 - C/Java/Rust: `while (C) { S }`
 - Python: `while C: \tab S`
 - **abstract syntax:** `while(C,S)`

The Programming Language L0

- Abstract Syntax (given by a **constructor signature**)

```
num:    int  $\rightarrow$  L0
bool:   boolean  $\rightarrow$  L0
id:     string  $\rightarrow$  L0
add:    L0  $\times$  L0  $\rightarrow$  L0
mul:    L0  $\times$  L0  $\rightarrow$  L0
....
if:      L0  $\times$  L0  $\rightarrow$  L0
let:     list[string  $\times$  L0]  $\times$  L0  $\rightarrow$  L0
fun:     string  $\times$  L0  $\rightarrow$  L0
app:     L0  $\times$  L0  $\rightarrow$  L0
```

The Programming Language L0

- Abstract Syntax (given by a **constructor signature**)

num: **int** \rightarrow L0
bool: **boolean** \rightarrow L0
id: **string** \rightarrow L0
add: L0 \times L0 \rightarrow L0
mul: L0 \times L0 \rightarrow L0
....
if: L0 \times L0 \rightarrow L0
let: list[**string** \times L0] \times L0 \rightarrow L0
fun: **string** \times L0 \rightarrow L0
app: L0 \times L0 \rightarrow L0

M, N (*Terms*) ::=

| x (*variable*)

| $\lambda x. M$ (*abstraction*)

| MN (*application*)



Naming

- Abstract Syntax (given by a **constructor signature**)

```
num:    int  $\rightarrow$  L0
bool:    boolean  $\rightarrow$  L0
id:      string  $\rightarrow$  L0
add:     L0  $\times$  L0  $\rightarrow$  L0
mul:     L0  $\times$  L0  $\rightarrow$  L0
....
if:      L0  $\times$  L0  $\rightarrow$  L0
let:     list[string  $\times$  L0]  $\times$  L0  $\rightarrow$  L0
fun:     string  $\times$  L0  $\rightarrow$  L0
app:     L0  $\times$  L0  $\rightarrow$  L0
```

Naming

- Names are the first abstraction mechanism in a programming language (and any language in fact!). Names allows us to refer to complex things, using **definitions**.
- A name used in some expression or program always denotes a value previously defined (**define before use**).
- Fundamentally, the meaning of a program fragment with names is obtained by **replacing** each name with the value assigned to it in its definition.
- **Key concepts related to naming:**
 - **Literals** versus **Identifiers** (technical jargon for “names”)
 - **Binding** (declaration) of names
 - **Scope** of a Definition
 - Occurrences of **Identifiers** (free, bound, binding)
 - Open and Closed code fragments

Literals vs Identifiers

- **Literals**

Denote fixed values in every context of occurrence

Java: true, false, "foo", float

Python: True, False, []

C: 1, 1.0, 0xFF, "hello", int

- **Identifiers**

Denote values that depend of the context of occurrence

In programming languages, identifiers are names for defined constants, variables, functions, methods, classes, modules, types, etc...

Java: x2, y, Count, System.out

C: printf

Binding and Scope

- **Binding**

The **association** between an identifier x and the value V it denotes is called a **binding** (of V to x).

- **Scope**

A binding for an identifier to the value associated is always established in a well-defined **static syntactical context** (a zone of the program **source text**)

The binding is created by a program construct called a **declaration**

The syntactical context (zone of the program text) in which the binding is established is called the **scope** of the binding / declaration.

Binding and Scope (examples)

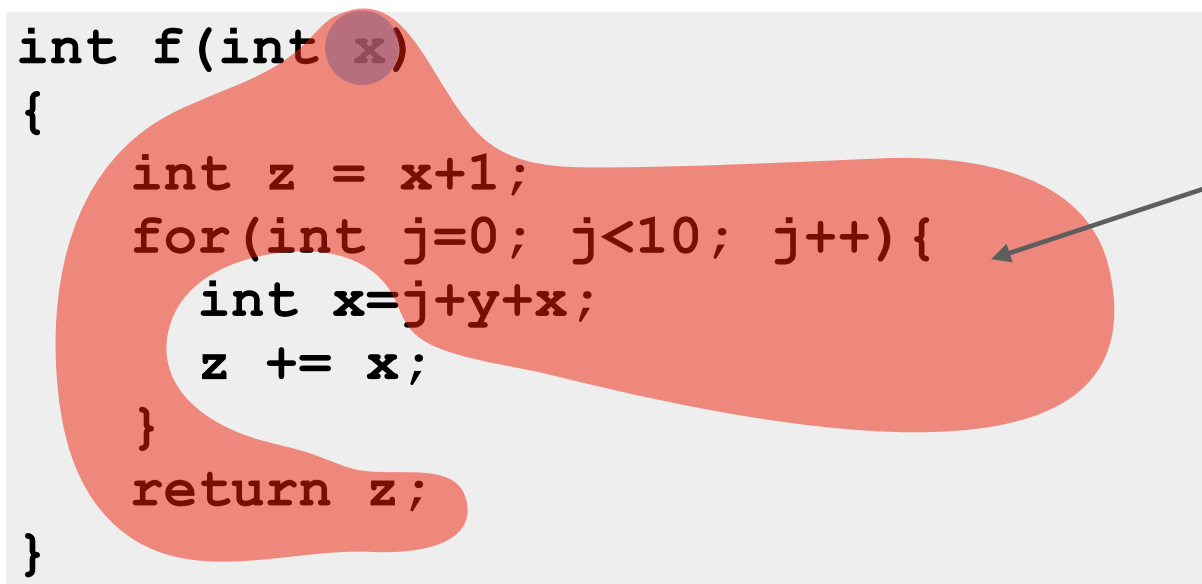
- The identifier `x` denotes (the address of) a memory cell

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y+x;
        z += x;
    }
    return z;
}
```

Binding and Scope (examples)

- The identifier `x` denotes (the address of) a memory cell

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y+x;
        z += x;
    }
    return z;
}
```



Scope of the Binding

Binding and Scope (examples)

- The identifier `j` denotes (the address of) a memory cell

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y+x;
        z += x;
    }
    return z;
}
```

Binding and Scope (examples)

- The identifier `j` denotes (the address of) a memory cell

```
int f(int x)
```

```
{
```

```
    int z = x+1;
```

```
    for(int j=0; j<10; j++) {
```

```
        int x=j+y+x;
```

```
        z += x;
```

```
    }
```

```
    return z;
```

```
}
```

Scope of the Binding



Components of a Scope

- The binding of an identifier X to its denotation V (value, memory address, function, type) **always involves** the following ingredients:
 - **A (single!) binding occurrence of the identifier X**
In general, it corresponds to the part of the program text that initialises the binding, where the binding becomes active
 - **The scope of the binding**
This is the part (zone of the program text) in which the binding introduced by the binding occurrence is active
 - **Several bound occurrences**
All occurrences of X , distinct from the binding occurrence, that lie inside the scope

Binding and Bound Occurrences

- Occurrences of identifier x

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y+x;
        z += x;
    }
    return z;
}
```

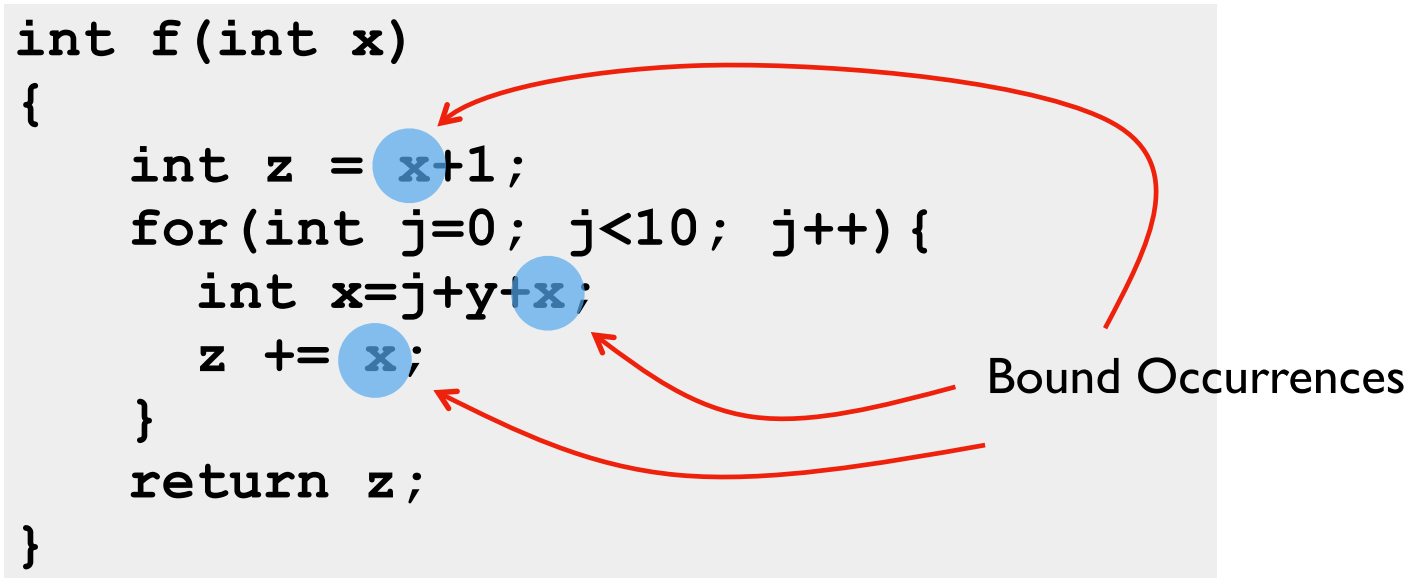
Binding Occurrences

Binding and Bound Occurrences

- Occurrences of identifier x

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y+x;
        z += x;
    }
    return z;
}
```

Bound Occurrences



Bound Occurrences

- For each bound occurrence there is exactly one binding occurrence (the one introduced in the declaration)

```
int f(int x)  
{  
    int z = x+1;  
    for(int j=0; j<10; j++){  
        int x=j+y+x;  
        z += x;  
    }  
    return z;  
}
```

Bound Occurrences

- For each bound occurrence there is exactly one binding occurrence (the one introduced in the declaration)

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++) {
        int x=j+y+x;
        z += x;
    }
    return z;
}
```

Bound Occurrences

- For each bound occurrence there is exactly one binding occurrence (the one introduced in the declaration)

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++) {
        int x=j+y+x;
        z += x;
    }
    return z;
}
```


Bound Occurrences

- For each bound occurrence there is exactly one binding occurrence (the one introduced in the declaration)

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++) {
        int x=j+y+x;
        z += x;
    }
    return z;
}
```

Free Occurrences

- Any occurrence of an identifier that is not binding nor bound in the program fragment is said to be **free** in the program fragment

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y+x;
        z += x;
    }
    return z;
}
```

- A binding for **y** must be provided by the outside program context.
NB. **y** is the only free identifier in the fragment shown.

Open and Closed Fragments

- A program fragment is said to be **open**
 - if it contains free occurrences of identifiers
- Otherwise, a program fragment is said to be **closed**
 - that is, it does not contain free occurrences of identifiers
- Examples of open program fragments

```
void f(int x)
{
    int i;
    for(int i=0;i<TEN;i++) x+=i;
    printf("%d\n",x);
}
```

C code

```
let x=1; (f x)
```

L0 code

The Language L0

$x, y, z \in \text{Var}$

$M, N \text{ (Terms)} ::=$

| x (variable)

| $\lambda x . M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

Variable Occurrences

$x, y, z \in \text{Var}$

M, N (Terms) ::=

| x (variable)

| $\lambda x . M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

We define the set **vars**(M) of **variables** in term M

$$\text{vars}(x) = \{x\}$$

$$\text{vars}(\lambda x . M) = \text{vars}(M) \cup \{x\}$$

$$\text{vars}(MN) = \text{vars}(M) \cup \text{vars}(N)$$

$$\text{vars}(n) = \emptyset \quad \text{vars}(b) = \emptyset$$

$$\text{vars}(M \text{ op } N) = \text{vars}(M) \cup \text{vars}(N)$$

$$\text{vars}(\text{if } M \text{ then } N \text{ else } R) = \text{vars}(M) \cup \text{vars}(N) \cup \text{vars}(R)$$

$$\text{vars}(\text{let } x = N \text{ in } M) = \text{vars}(N) \cup \text{vars}(M) \cup \{x\}$$

Free Variables

$x, y, z \in \text{Var}$

M, N (Terms) ::=

| x (variable)

| $\lambda x . M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

In the λ -calculus, variables are mostly “declared” as function parameters and “used” in function bodies

All other variables are called **free variables**

We define the set **fv**(M) of **free variables** of term M by

$$\text{fv}(x) = \{x\}$$

$$\text{fv}(\lambda x . M) = \text{fv}(M) \setminus \{x\}$$

$$\text{fv}(MN) = \text{fv}(M) \cup \text{fv}(N)$$

Free Variables

$x, y, z \in \text{Var}$

M, N (Terms) ::=

| x (variable)

| $\lambda x . M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

In the λ -calculus, variables are mostly “declared” as function parameters and “used” in function bodies

All other variables are called **free variables**


We define the set **fv**(M) of **free variables** of term M by

$$\text{fv}(x) = \{x\}$$

$$\text{fv}(\lambda x . M) = \text{fv}(M) \setminus \{x\}$$

$$\text{fv}(MN) = \text{fv}(M) \cup \text{fv}(N)$$

all free occurrences
of x in M
become **bound** in
 $\lambda x . M$



Free Variables

$x, y, z \in \text{Var}$

M, N (Terms) ::=

| x (variable)

| $\lambda x . M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

In the λ -calculus, variables are mostly “declared” as function parameters and “used” in function bodies

All other variables are called **free variables**

We define the set **fv**(M) of **free variables** of term M by

$$\text{fv}(x) = \{x\}$$

$$\text{fv}(\lambda x . M) = \text{fv}(M) \setminus \{x\}$$

$$\text{fv}(MN) = \text{fv}(M) \cup \text{fv}(N)$$

all occurrences in M
of a variable x which is not free in M
are said to be **bound** occurrences

Free Variables

$x, y, z \in \text{Var}$

M, N (Terms) ::=

| x (variable)

| $\lambda x. M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

In the λ -calculus, variables are mostly “declared” as function parameters and “used” in function bodies

All other variables are called **free variables**

We define the set **fv**(M) of **free variables** of term M by

$$\text{fv}(n) = \emptyset \quad \text{fv}(b) = \emptyset$$

$$\text{fv}(M \text{ op } N) = \text{fv}(M) \cup \text{fv}(N)$$

$$\text{fv}(\text{if } M \text{ then } N \text{ else } R) = \text{fv}(M) \cup \text{fv}(N) \cup \text{fv}(R)$$

$$\text{fv}(\text{let } x = N \text{ in } M) = \text{fv}(N) \cup (\text{fv}(M) \setminus \{x\})$$

Free Variables

$x, y, z \in \text{Var}$

$M, N \text{ (Terms)} ::=$

| x (variable)

| $\lambda x. M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

Examples:

fv($\lambda x. (x + y)$) =

fv($\lambda x. (f x) y$) =

fv ($\lambda x. (x y) (\lambda y. (x y))$) =

fv ($\lambda y. (\lambda x. (x y) (\lambda y. (x y)))$) =

fv (let $x = 42 + z$ in $x + y$) =

Substitution

$x, y, z \in \text{Var}$

$M, N \text{ (Terms)} ::=$

| x (variable)

| $\lambda x . M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

In a λ -calculus, **binding** of variables to values is modelled by **syntactic substitution**.

We denote by $\{N/x\}M$ the term obtained by substituting all **free** occurrences of x in M by N .

$$N = \lambda x . (a \ x)$$

$$\{x/N\}(x \ x \ y) = (N \ N \ y) = ((\lambda x . (a \ x)) (\lambda x . (a \ x)) \ y)$$

$$\{x/N\}\lambda z . (x \ z) = \lambda z . (N \ z) = \lambda z . ((\lambda x . (a \ x)) \ z)$$

$$\{x/N\}\lambda a . (x \ a) = \lambda a . (N \ a) = \lambda a . ((\lambda x . (a \ x)) \ a)$$

Substitution

$x, y, z \in \text{Var}$

$M, N \text{ (Terms)} ::=$

| x (variable)

| $\lambda x . M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

In a λ -calculus, **binding** of variables to values is modelled by **syntactic substitution**.

We denote by $\{N/x\}M$ the term obtained by substituting all **free** occurrences of x in M by N .

$$N = \lambda x . (a \ x)$$

$$\{x/N\}(x \ x \ y) = (N \ N \ y) = ((\lambda x . (a \ x)) (\lambda x . (a \ x)) \ y)$$

$$\{x/N\}\lambda z . (x \ z) = \lambda z . (N \ z) = \lambda z . ((\lambda x . (a \ x)) \ z)$$

$$\{x/N\}\lambda a . (x \ a) = \lambda a . (N \ a) = \lambda a . ((\lambda x . (a \ x)) \ a)$$

incorrect capture of free a in N

Substitution

$x, y, z \in \text{Var}$

$M, N \text{ (Terms)} ::=$

| x (variable)

| $\lambda x . M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

In a λ -calculus, **binding** of variables to values is modelled by **syntactic substitution**.

We denote by $\{N/x\}M$ the term obtained by substituting all **free** occurrences of x in M by N .

$$N = \lambda x . (a \ x)$$

$$\{x/N\}(x \ x \ y) = (N \ N \ y) = ((\lambda x . (a \ x)) (\lambda x . (a \ x)) \ y)$$

$$\{x/N\}\lambda z . (x \ z) = \lambda z . (N \ z) = \lambda z . ((\lambda x . (a \ x)) \ z)$$

$$\{x/N\}\lambda a . (f \ a) = \boxed{\{x/N\}\lambda a' . (f \ a')} = \lambda a' . ((\lambda x . (a \ x)) \ a')$$

rename bound variable to avoid capture!

α -equivalence

$x, y, z \in \text{Var}$

$M, N \text{ (Terms)} ::=$

$| x \text{ (variable)}$

$| \lambda x . M \text{ (abstraction)}$

$| MN \text{ (application)}$

$| n \text{ (integer)}$

$| M \text{ op } M \text{ (operation)}$

$| \text{if } M \text{ then } N \text{ else } R \text{ (conditional)}$

In a λ -calculus, α -equivalence (“alphabetic” equivalence) identifies terms that only differ by clash-free renaming of bound variables.

It models the fact that choices of local variable names or parameter names in programs do not change meaning.

```
succ = lambda a: a+1  
succ = lambda x: x+1
```

α -equivalent definitions

law of α -equivalence

$$\lambda x . M =_{\alpha} \lambda z . \{z/x\}M \quad (z \notin \text{vars}(M))$$

α -equivalence

$x, y, z \in \text{Var}$

M, N (Terms) ::=

| x (variable)

| $\lambda x. M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

In a λ -calculus, α -equivalence (“alphabetic” equivalence) identifies terms that only differ by clash-free renaming of bound variables.

It models the fact that choices of local variable names or parameter names in programs do not change meaning.

```
def flip(x,y):  
    return (y,x)  
def flip(a,b):  
    return (b,a)
```

α -equivalent definitions

law of α -equivalence

$$\lambda x. M =_{\alpha} \lambda z. \{z/x\}M \quad (z \notin \text{vars}(M))$$

Substitution

$x, y, z \in \text{Var}$

$M, N \text{ (Terms)} ::=$

| x (variable)

| $\lambda x . M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

In a λ -calculus, **binding** of variables to values is modelled by **syntactic substitution**.

We denote by $\{N/x\}M$ the term obtained by substituting all **free** occurrences of x in M by N .

$$N = \lambda x . (a \ x)$$

$$\{x/N\}(x \ x \ y) = (N \ N \ y) = ((\lambda x . (a \ x)) (\lambda x . (a \ x)) \ y)$$

$$\{x/N\}\lambda z . (x \ z) = \lambda z . (N \ z) = \lambda z . ((\lambda x . (a \ x)) \ z)$$

$$\{x/N\}\lambda a . (f \ a) =_{\alpha} \{x/N\}\lambda a' . (f \ a') = \lambda a' . ((\lambda x . (a \ x)) \ a')$$


use of α -equivalence

Safe Substitution

Safe substitution assumes that bound variables are always distinct from free variables, implicitly applying α -equivalence renamings (the so-called “Barendregt convention”).

Capture avoiding safe substitution is inductively defined as follows:

$$\begin{aligned} \{N/x\}x &= N & \{N/x\}y &= y \quad (x \neq y) & \{N/x\}n &= n & \{N/x\}b &= b \\ \{N/x\}\lambda z. M &= \lambda z. \{N/x\}M \quad (z \notin \text{fv}(M)) & \{R/x\}(MN) &= (\{R/x\}M)\{R/x\}N \\ \{R/x\}(\text{let } z = N \text{ in } M) &= \text{let } z = \{R/x\}N \text{ in } \{R/x\}M \quad (z \notin \text{fv}(M)) \\ \{R/x\}(M \text{ op } N) &= \{R/x\}M \text{ op } \{R/x\}N \\ \{R/x\}\text{if } M \text{ then } N \text{ else } S &= \text{if } \{R/x\}M \text{ then } \{R/x\}N \text{ else } \{R/x\}S \end{aligned}$$

Big-Step Semantics

The semantics of a L0 program
may also be given by a so-called (big-step) **evaluation** relation

$$M \Downarrow N \quad (M \text{ evaluates to value } N)$$

Big-step semantics directly defines the evaluation from program to final value (if any)

We have $M \Downarrow N$ **if and only if** $M \rightarrow \cdot \rightarrow \dots \rightarrow N$ and N is a value

Values

Values are expressions V that do not reduce and represent the simplest form to refer to the result of a computation (also referred to **canonical** or **normal forms**).

We consider the **values** of L0 to be the **integers**, **booleans** and **function objects**

V, U (*Values*) ::=

| n (*integer*)

| b (*boolean*)

| $\lambda x. M$ (*abstraction*)

We will write

value M

to assert that term M is a value

A reduction semantics usually imposes some order of evaluation towards producing a final value, called a **reduction strategy**

Call-by-Value Big-Step Semantics

Instead of defining single reduction steps, one may define a direct evaluation relation, from program to result, called **big-step reduction semantics**.

$$\frac{M \text{ value}}{M \Downarrow M}$$

$$\frac{N \Downarrow \lambda x. R \quad M \Downarrow V \quad \{V/x\}R \Downarrow U}{NM \Downarrow U}$$

$$\frac{M \Downarrow n \quad N \Downarrow k \quad \text{add}(n, k) = r}{M + N \Downarrow r}$$

$$\frac{N \Downarrow V \quad \{x/V\}M \Downarrow U}{\text{let } x = N \text{ in } M \Downarrow U}$$

$$\frac{M \Downarrow \text{true} \quad N \Downarrow V}{\text{if } M \text{ then } N \text{ else } M \Downarrow V}$$

$$\frac{M \Downarrow \text{false} \quad R \Downarrow V}{\text{if } M \text{ then } N \text{ else } R \Downarrow V}$$

Call-by-Value Big-Step Semantics

Instead of defining single reduction steps, one may define a direct evaluation relation, from program to result, called **big-step reduction semantics**.

$$\begin{array}{c}
 \frac{M \text{ value}}{M \Downarrow M} \qquad \text{one proof rule} \qquad \frac{N \Downarrow \lambda x. R \quad M \Downarrow V \quad \{V/x\}R \Downarrow U}{NM \Downarrow U} \\
 \\
 \frac{M \Downarrow n \quad N \Downarrow k \quad \text{add}(n, k) = r}{M + N \Downarrow r} \qquad \boxed{\frac{N \Downarrow V \quad \{x/V\}M \Downarrow U}{\text{let } x = N \text{ in } M \Downarrow U}} \\
 \\
 \frac{M \Downarrow \text{true} \quad N \Downarrow V}{\text{if } M \text{ then } N \text{ else } M \Downarrow V} \qquad \frac{M \Downarrow \text{false} \quad R \Downarrow V}{\text{if } M \text{ then } N \text{ else } R \Downarrow V}
 \end{array}$$

Call-by-Value Big-Step Semantics

Instead of defining single reduction steps, one may define a direct evaluation relation, from program to result, called **big-step reduction semantics**.

$$\begin{array}{c}
 \frac{M \text{ value}}{M \Downarrow M} \qquad \text{proof rule premise} \qquad \frac{N \Downarrow \lambda x. R \quad M \Downarrow V \quad \{V/x\}R \Downarrow U}{NM \Downarrow U} \\
 \\
 \frac{M \Downarrow n \quad N \Downarrow k \quad \text{add}(n, k) = r}{M + N \Downarrow r} \qquad \frac{\boxed{N \Downarrow V} \quad \{x/V\}M \Downarrow U}{\text{let } x = N \text{ in } M \Downarrow U} \\
 \\
 \frac{M \Downarrow \text{true} \quad N \Downarrow V}{\text{if } M \text{ then } N \text{ else } M \Downarrow V} \qquad \frac{M \Downarrow \text{false} \quad R \Downarrow V}{\text{if } M \text{ then } N \text{ else } R \Downarrow V}
 \end{array}$$

Call-by-Value Big-Step Semantics

Instead of defining single reduction steps, one may define a direct evaluation relation, from program to result, called **big-step reduction semantics**.

$$\frac{M \text{ value}}{M \Downarrow M}$$

$$\frac{N \Downarrow \lambda x. R \quad M \Downarrow V \quad \{V/x\}R \Downarrow U}{NM \Downarrow U}$$

$$\frac{M \Downarrow n \quad N \Downarrow k \quad \text{add}(n, k) = r}{M + N \Downarrow r}$$

$$N \Downarrow V \quad \{x/V\}M \Downarrow U$$

$$\text{let } x = N \text{ in } M \Downarrow U$$

proof rule conclusion

$$\frac{M \Downarrow \text{true} \quad N \Downarrow V}{\text{if } M \text{ then } N \text{ else } M \Downarrow V}$$

$$\frac{M \Downarrow \text{false} \quad R \Downarrow V}{\text{if } M \text{ then } N \text{ else } R \Downarrow V}$$

Call-by-Value Big-Step Semantics

Instead of defining single reduction steps, one may define a direct evaluation relation, from program to result, called **big-step reduction semantics**.

value judgement

$$\boxed{M \text{ value}} \quad M \Downarrow M$$

$$\frac{N \Downarrow \lambda x. R \quad M \Downarrow V \quad \{V/x\}R \Downarrow U}{NM \Downarrow U}$$

$$\frac{M \Downarrow n \quad N \Downarrow k \quad \text{add}(n, k) = r}{M + N \Downarrow r}$$

$$\frac{N \Downarrow V \quad \{x/V\}M \Downarrow U}{\text{let } x = N \text{ in } M \Downarrow U}$$

$$\frac{M \Downarrow \text{true} \quad N \Downarrow V}{\text{if } M \text{ then } N \text{ else } M \Downarrow V}$$

$$\frac{M \Downarrow \text{false} \quad R \Downarrow V}{\text{if } M \text{ then } N \text{ else } R \Downarrow V}$$

Call-by-Value Big-Step Semantics

Instead of defining single reduction steps, one may define a direct evaluation relation, from program to result, called **big-step reduction semantics**.

$$\begin{array}{c}
 \frac{M \text{ value}}{M \Downarrow M} \\
 \\
 \frac{M \Downarrow n \quad N \Downarrow k \quad \boxed{\text{add}(n, k) = r}}{M + N \Downarrow r} \quad \begin{array}{l} \text{judgement to} \\ \text{perform aux} \\ \text{computation} \\ \text{(addition here)} \end{array} \\
 \\
 \frac{N \Downarrow \lambda x. R \quad M \Downarrow V \quad \{V/x\}R \Downarrow U}{NM \Downarrow U} \\
 \\
 \frac{N \Downarrow V \quad \{x/V\}M \Downarrow U}{\text{let } x = N \text{ in } M \Downarrow U} \\
 \\
 \frac{M \Downarrow \text{true} \quad N \Downarrow V}{\text{if } M \text{ then } N \text{ else } M \Downarrow V} \quad \frac{M \Downarrow \text{false} \quad R \Downarrow V}{\text{if } M \text{ then } N \text{ else } R \Downarrow V}
 \end{array}$$

Call-by-Value Big-Step Proof Tree

$$\frac{3 \Downarrow 3 \quad \text{let } f = \lambda z. 3 * z ; (f (2 + 3)) \Downarrow 15}{\text{let } k = 3; \text{let } f = \lambda z. k * z ; (f (2 + k)) \Downarrow 15}$$

$$\frac{\{3/k\} \text{let } k = 3; \text{let } f = \lambda z. k * z ; (f (2 + k))}{\text{let } f = \lambda z. 3 * z ; (f (2 + 3))}$$

Call-by-Value Big-Step Proof Tree

$$\frac{\lambda z.3 * z \Downarrow \lambda z.3 * z \quad (\lambda z.3 * z)(2 + 3) \Downarrow 15}{\text{let } f = \lambda z.3 * z ; (f (2 + 3)) \Downarrow 15}$$

$$\frac{3 \Downarrow 3 \quad \text{let } f = \lambda z.3 * z ; (f (2 + 3)) \Downarrow 15}{\text{let } k = 3; \text{let } f = \lambda z.k * z ; (f (2 + k)) \Downarrow 15}$$

$$\begin{aligned} & \{(\lambda z.3 * z)/f\}(f(2 + 3)) \\ & \quad = \\ & (\lambda z.3 * z)(2 + 3) \end{aligned}$$

$$\begin{aligned} & \{3/k\} \text{let } k = 3; \text{let } f = \lambda z.k * z ; (f (2 + k)) \\ & \quad = \\ & \text{let } f = \lambda z.3 * z ; (f (2 + 3)) \end{aligned}$$

Call-by-Value Big-Step Proof Tree

$$\frac{3 \Downarrow 3 \quad 5 \Downarrow 5 \quad \text{add}(3,5) = 15}{3 * 5 \Downarrow 15}$$

$$\frac{2 + 3 \Downarrow 5 \quad 3 * 5 \Downarrow 15}{(\lambda z.3 * z)(2 + 3) \Downarrow 15}$$

$$\frac{\lambda z.3 * z \Downarrow \lambda z.3 * z \quad (\lambda z.3 * z)(2 + 3) \Downarrow 15}{\text{let } f = \lambda z.3 * z ; (f (2 + 3)) \Downarrow 15}$$

$$\frac{3 \Downarrow 3 \quad \text{let } f = \lambda z.3 * z ; (f (2 + 3)) \Downarrow 15}{\text{let } k = 3; \text{let } f = \lambda z.k * z ; (f (2 + k)) \Downarrow 15}$$

$$\begin{aligned} & \{(\lambda z.3 * z)/f\}(f(2 + 3)) \\ & \quad = \\ & (\lambda z.3 * z)(2 + 3) \end{aligned}$$

$$\begin{aligned} & \{3/k\} \text{let } k = 3; \text{let } f = \lambda z.k * z ; (f (2 + k)) \\ & \quad = \\ & \text{let } f = \lambda z.3 * z ; (f (2 + 3)) \end{aligned}$$

Call-by-Value Big-Step Proof Tree

$$\frac{3 \Downarrow 3 \quad 5 \Downarrow 5 \quad \text{add}(3,5) = 15}{3 * 5 \Downarrow 15}$$

$$\frac{2 + 3 \Downarrow 5 \quad 3 * 5 \Downarrow 15}{(\lambda z.3 * z)(2 + 3) \Downarrow 15}$$

$$\frac{\lambda z.3 * z \Downarrow \lambda z.3 * z \quad (\lambda z.3 * z)(2 + 3) \Downarrow 15}{\text{let } f = \lambda z.3 * z ; (f (2 + 3)) \Downarrow 15}$$

$$\frac{3 \Downarrow 3 \quad \text{let } f = \lambda z.3 * z ; (f (2 + 3)) \Downarrow 15}{\text{let } k = 3; \text{let } f = \lambda z.k * z ; (f (2 + k)) \Downarrow 15}$$

$$\begin{aligned} & \{(\lambda z.3 * z)/f\}(f(2 + 3)) \\ & \quad = \\ & (\lambda z.3 * z)(2 + 3) \end{aligned}$$

$$\begin{aligned} & \{3/k\} \text{let } k = 3; \text{let } f = \lambda z.k * z ; (f (2 + k)) \\ & \quad = \\ & \text{let } f = \lambda z.3 * z ; (f (2 + 3)) \end{aligned}$$

Call-by-Value Big-Step Proof Tree

$$\begin{array}{c}
 \frac{3 \Downarrow 3 \quad 5 \Downarrow 5 \quad \text{add}(3,5) = 15}{3 * 5 \Downarrow 15} \\
 \frac{2 + 3 \Downarrow 5 \quad 3 * 5 \Downarrow 15}{(\lambda z.3 * z)(2 + 3) \Downarrow 15} \\
 \frac{\lambda z.3 * z \Downarrow \lambda z.3 * z \quad (\lambda z.3 * z)(2 + 3) \Downarrow 15}{\text{let } f = \lambda z.3 * z ; (f (2 + 3)) \Downarrow 15} \\
 \frac{3 \Downarrow 3 \quad \text{let } f = \lambda z.3 * z ; (f (2 + 3)) \Downarrow 15}{\text{let } k = 3; \text{let } f = \lambda z.k * z ; (f (2 + k)) \Downarrow 15}
 \end{array}$$

$$\begin{array}{c}
 \{(\lambda z.3 * z)/f\}(f(2 + 3)) \\
 = \\
 (\lambda z.3 * z)(2 + 3) \\
 \\
 \{3/k\}\text{let } k = 3; \text{let } f = \lambda z.k * z ; (f (2 + k)) \\
 = \\
 \text{let } f = \lambda z.3 * z ; (f (2 + 3))
 \end{array}$$

Call-by-Value Big-Step Proof Tree

$$\begin{array}{c}
 \frac{3 \Downarrow 3 \quad 5 \Downarrow 5 \quad \text{add}(3,5) = 15}{2 + 3 \Downarrow 5 \quad 3 * 5 \Downarrow 15} \\
 \frac{\lambda z.3 * z \Downarrow \lambda z.3 * z \quad (\lambda z.3 * z)(2 + 3) \Downarrow 15}{3 \Downarrow 3 \quad \text{let } f = \lambda z.3 * z ; (f (2 + 3)) \Downarrow 15} \\
 \hline
 \text{let } k = 3; \text{let } f = \lambda z.k * z ; (f (2 + k)) \Downarrow 15
 \end{array}$$

Environment Semantics

Substitutions are convenient to easily model evaluation and reason about program behaviour using algebraic manipulation, but are inadequate for implementation.

We will now introduce an **equivalent alternative semantics** more machine-oriented, that instead of substitutions relies on **environments**

The key idea is to avoid implementing substitutions of variables for terms and keep a structured dictionary that associates free ids of terms to their values

Technically, such dictionary is called an **environment**, and is conceptually a set of assignments of values to ids

A program is always executed w.r.t. an environment, storing the values of its free ids. Whenever a id is evaluated, its value is fetched from the environment

Environment Semantics for L0

$x, y, z \in \text{Var}$

M, N (Terms) ::=

| x (variable)

| $\lambda x . M$ (abstraction)

| MN (application)

| b (boolean)

| n (integer)

| $M \text{ op } M$ (operation)

| if M then N else R (conditional)

| let $x = N$ in M (definition)

V, U (Values) ::=

| n (integer)

| $\text{clos}(\mathcal{E}, \lambda x . M)$ (closure)

\mathcal{E}, \mathcal{F} (Environments) :

| \emptyset (empty)

| $\mathcal{E}[x \mapsto V]$ (binding)

Operations on Environments

The empty environment

\emptyset (*empty*)

evaluation of programs (**closed terms**) always starts with \emptyset

Environment update

$\mathcal{E}[x \mapsto V]$ (*binding*)

updates environment with new binding (of V to x)

Environment lookup

$\mathcal{E}(x)$ (*lookup x*)

lookup of value associated to x in \mathcal{E}

$$\mathcal{E}[y \mapsto V](x) = \begin{cases} V & \text{if } x = y \\ \mathcal{E}(x) & \text{if } x \neq y \end{cases}$$

Big Step Environment Semantics

$\mathcal{E}; M \Downarrow N$ (M evaluates to N in environment \mathcal{E})

$$\frac{n \text{ integer}}{\mathcal{E}; n \Downarrow n}$$

$$\frac{n \text{ boolean}}{\mathcal{E}; n \Downarrow n}$$

$$\frac{}{\mathcal{E}; x \Downarrow \mathcal{E}(x)}$$

$$\frac{}{\mathcal{E}; \lambda x. M \Downarrow \text{clos}(\mathcal{E}, \lambda x. M)}$$

$$\frac{\mathcal{E}; N \Downarrow \text{clos}(\mathcal{E}, \lambda x. R) \quad \mathcal{E}; M \Downarrow V \quad \mathcal{E}[x \mapsto V]; R \Downarrow U}{\mathcal{E}; NM \Downarrow U}$$

$$\frac{\mathcal{E}; M \Downarrow n \quad \mathcal{E}; N \Downarrow k \quad \text{add}(n, k) = r}{\mathcal{E}; M + N \Downarrow r}$$

$$\frac{\mathcal{E}; N \Downarrow V \quad \mathcal{E}[x \mapsto V]; M \Downarrow U}{\mathcal{E}; \text{let } x = N; M \Downarrow U}$$

$$\frac{\mathcal{E}; M \Downarrow \text{false} \quad \mathcal{E}; N \Downarrow V}{\mathcal{E}; \text{if } M \text{ then } N \text{ else } N \Downarrow V}$$

$$\frac{\mathcal{E}; M \Downarrow \text{true} \quad \mathcal{E}; R \Downarrow V}{\mathcal{E}; \text{if } M \text{ then } N \text{ else } R \Downarrow V}$$

Big Step Environment Semantics

$$\frac{\mathcal{E}_2; k \Downarrow 3 \quad \mathcal{E}_2; z \Downarrow 5 \quad \text{add}(3,5) = 15}{\mathcal{E}_2; k * z \Downarrow 15}$$

$$\frac{\mathcal{E}_1; f \Downarrow \text{clos}(\mathcal{E}_0, \lambda z. k * z) \quad \mathcal{E}_1; 2 * k \Downarrow 15 \quad \mathcal{E}_2; k * z \Downarrow 15}{\mathcal{E}_1; f(2 + k) \Downarrow 15}$$

$$\mathcal{E}_2 = [k \rightarrow 3][z \rightarrow 15]$$

$$\frac{\mathcal{E}_0; \lambda z. 3 * z \Downarrow \text{clos}(\mathcal{E}_0, \lambda z. k * z) \quad \mathcal{E}_1; f(2 + k) \Downarrow 15}{\mathcal{E}_0; \text{let } f = \lambda z. 3 * z ; (f(2 + k)) \Downarrow 15}$$

$$\mathcal{E}_1 = [k \rightarrow 3][f \rightarrow \text{clos}(\mathcal{E}_0, \lambda z. k * z)]$$

$$\frac{\emptyset; 3 \Downarrow 3 \quad \mathcal{E}_0; \text{let } f = \lambda z. 3 * z ; (f(2 + k)) \Downarrow 15}{\emptyset; \text{let } k = 3; \text{let } f = \lambda z. k * z ; (f(2 + k)) \Downarrow 15}$$

$$\mathcal{E}_0 = [k \rightarrow 3]$$

Big Step Environment Semantics

$$\mathcal{E}_2 = [k \rightarrow 3][z \rightarrow 15]$$

$$\mathcal{E}_1 = [k \rightarrow 3][f \rightarrow \text{clos}(\mathcal{E}_0, \lambda z. k * z)]$$

$$\mathcal{E}_0 = [k \rightarrow 3]$$

$$\frac{\frac{\frac{\frac{\mathcal{E}_0; \lambda z. 3 * z \Downarrow \text{clos}(\mathcal{E}_0, \lambda z. k * z)}{\mathcal{E}_1; f \Downarrow \text{clos}(\mathcal{E}_0, \lambda z. k * z)} \quad \mathcal{E}_1; 2 + k \Downarrow 5 \quad \mathcal{E}_2; k * z \Downarrow 15}{\mathcal{E}_1; f(2 + k) \Downarrow 15} \quad \mathcal{E}_2; k \Downarrow 3 \quad \mathcal{E}_2; z \Downarrow 15 \quad 3 * 15 = 15}{\frac{\emptyset; 3 \Downarrow 3 \quad \mathcal{E}_0; \text{let } f = \lambda z. 3 * z ; (f(2 + k)) \Downarrow 15}{\emptyset; \text{let } k = 3; \text{let } f = \lambda z. k * z ; (f(2 + k)) \Downarrow 15}}$$

Outline of an Interpreter for L0

```
public class L0int {  
  
    public static void main(String args[]) {  
        Parser parser = new Parser(System.in);  
        ASTNode exp;  
  
        System.out.println("L0 interpreter PL MEIC 2024/25 (v0.0)\n");  
  
        while (true) {  
            try {  
                System.out.print("# ");  
                exp = parser.Start();  
                if (exp==null) System.exit(0);  
                IValue v = exp.eval(new Environment<IValue>());  
                System.out.println(v.toStr());  
            } catch (ParseException e) {  
                System.out.println("Syntax Error.");  
                parser.ReInit(System.in);  
            }  
            catch (Exception e) {  
                e.printStackTrace();  
                parser.ReInit(System.in);  
            }  
        }  
    }  
}
```

Implementation of Environments

In practice, it is convenient to implement environments, using a mutable tree-like data structure called a “**spaghetti stack**”.

In usual block structured languages the addition and remotion of biddings between identifiers and values follows a strict stack LIFO discipline.

An environment stores all bindings relative to the current scope and all involving scopes in frames.

From any environment state one may create a new “child” frame, corresponding to a new nested scope.

Each frame links to the ancestor frame using a reference (dynamic link).

Environment operations

Environment **beginScope**(Environment e)

Pushes into the environment a new frame, where new bindings will be stored.

A given identifier can only be bound once in a given frame, but may be bound in different frames (to possibly different values).

Environment as an ADT

Environment **endScope**(Environment e)

Returns the parent environment of e (pops off the top stack frame)

The popped off frame is not deallocated (as it might be still referenced in closures)

Environment as an ADT

Environment **assoc**(Environment e, String id, IValue v)

Adds a new binding for identifier id to the value val in the top frame of the environment (if id is not bound there yet).

assoc may be implemented imperatively, the updated environment is e updated in place

Environment as an ADT

Value **find**(Environment e, String id)

Returns the value associated to id in the environment e, as defined by the innermost binding (the binding in the topmost frame that binds id).

In practice, find searches for id from top to bottom following the stack frame chain, from “most recent” up, so that the appropriate scoping is well-respected.

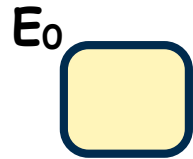
Example Evaluation #1

```
let a = 3 ;
```

```
let b = a * (let k = a + a; 2*k);
```

```
4*b
```

Example Evaluation #1

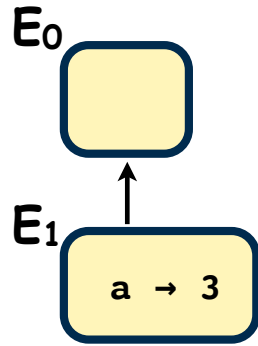


```
let a = 3 ;
```

```
let b = a * (let k = a + a; 2*k);
```

```
4*b
```

Example Evaluation #1

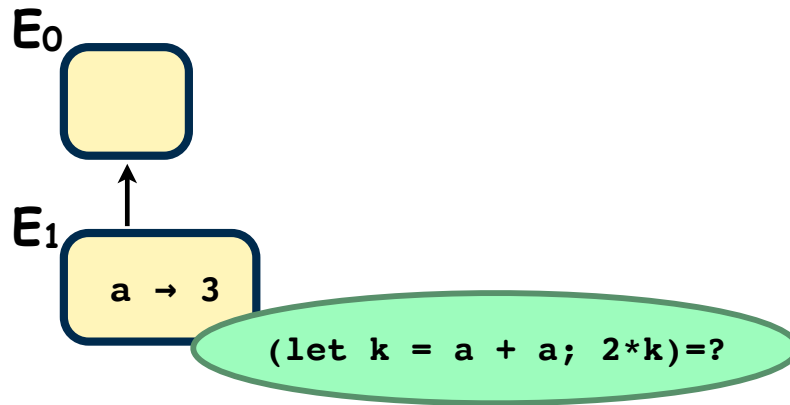


```
let a = 3 ;
```

```
let b = a * (let k = a + a; 2*k);
```

```
4*b
```

Example Evaluation #1

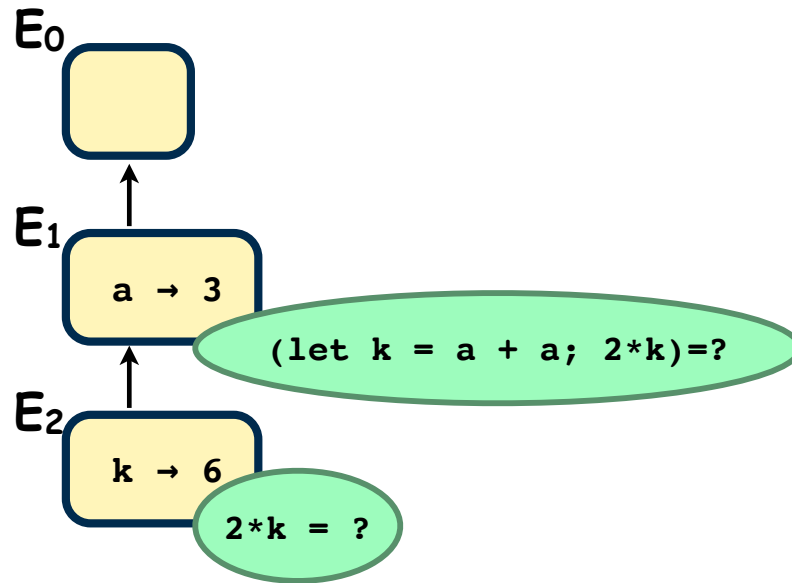


```
let a = 3 ;
```

```
let b = a * (let k = a + a; 2*k);
```

```
4*b
```

Example Evaluation #1

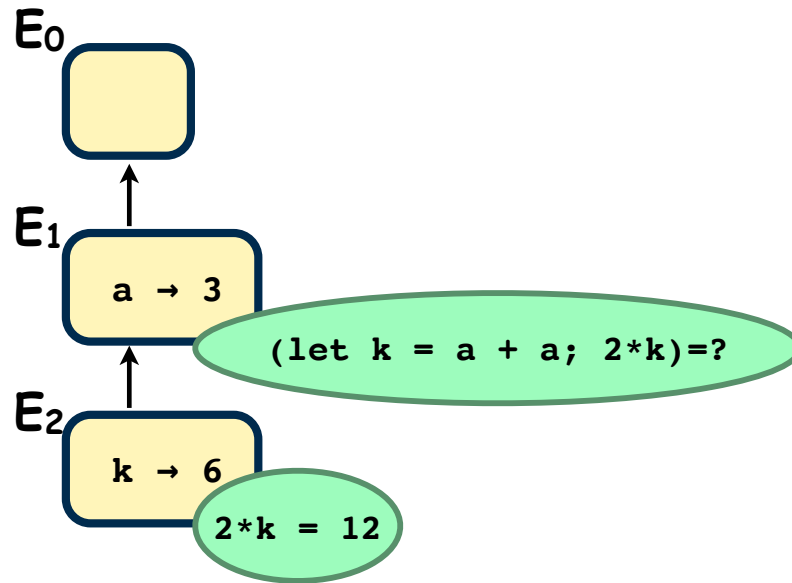


```
let a = 3 ;
```

```
let b = a * (let k = a + a; 2*k);
```

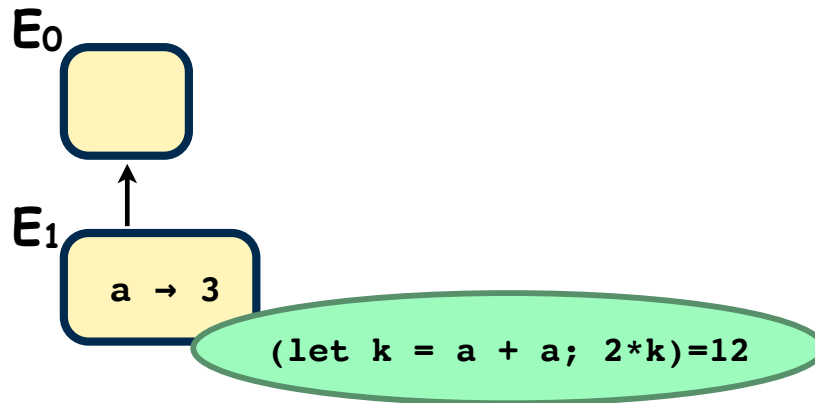
```
4*b
```


Example Evaluation #1



```
let a = 3 ;  
  
let b = a * (let k = a + a; 2*k);  
  
4*b
```

Example Evaluation #1

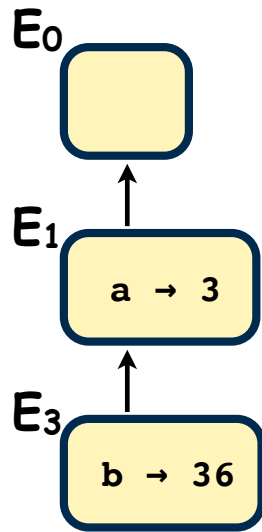


```
let a = 3 ;
```

```
let b = a * (let k = a + a; 2*k);
```

```
4*b
```

Example Evaluation #1

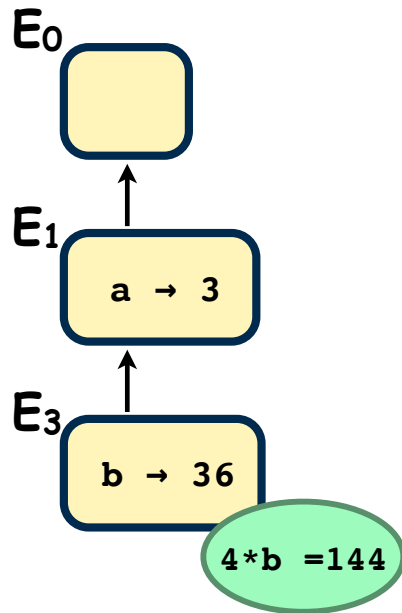


```
let a = 3 ;
```

```
let b = a * (let k = a + a; k) ;
```

```
4*b
```

Example Evaluation #1



```
let a = 3 ;
```

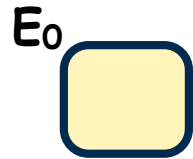
```
let b = a * (let k = a + a; k) ;
```

$4 * b$

Example Evaluation #2

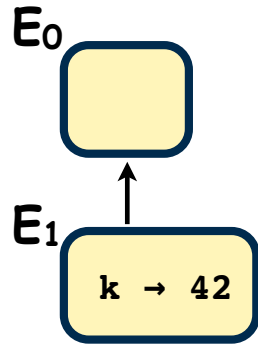
```
let k = 42 ;  
  
let f = fn y => { y+k } ;  
  
f(2)
```

Example Evaluation #2



```
let k = 42 ;  
  
let f = fn y => { y+k } ;  
  
f(2)
```

Example Evaluation #2

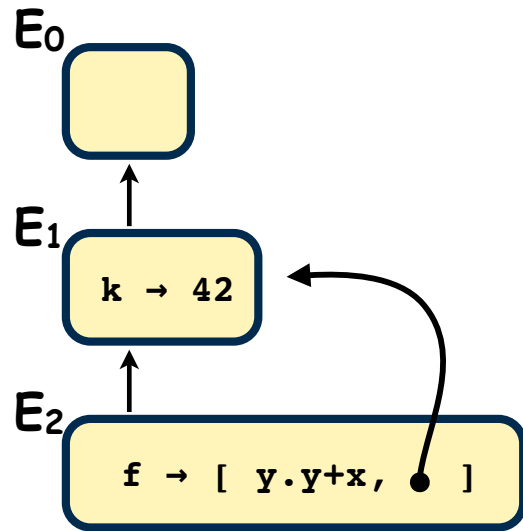


```
let k = 42 ;
```

```
let f = fn y => { y+k } ;
```

```
f(2)
```

Example Evaluation #2

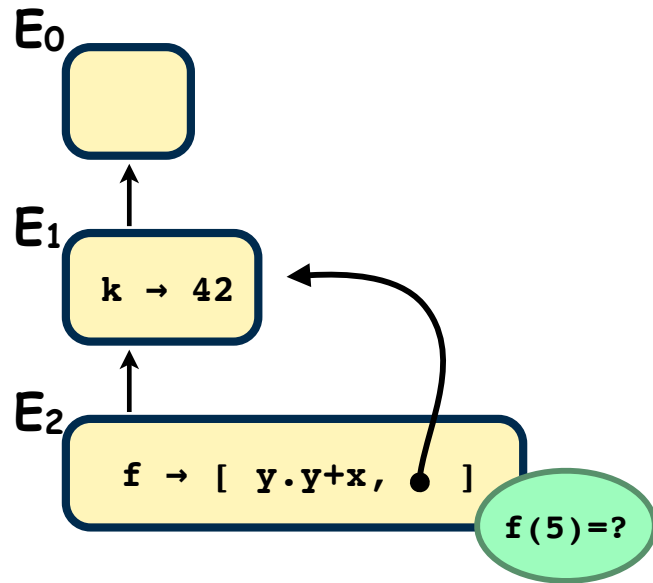


```
let k = 42 ;
```

```
let f = fn y => { y+k } ;
```

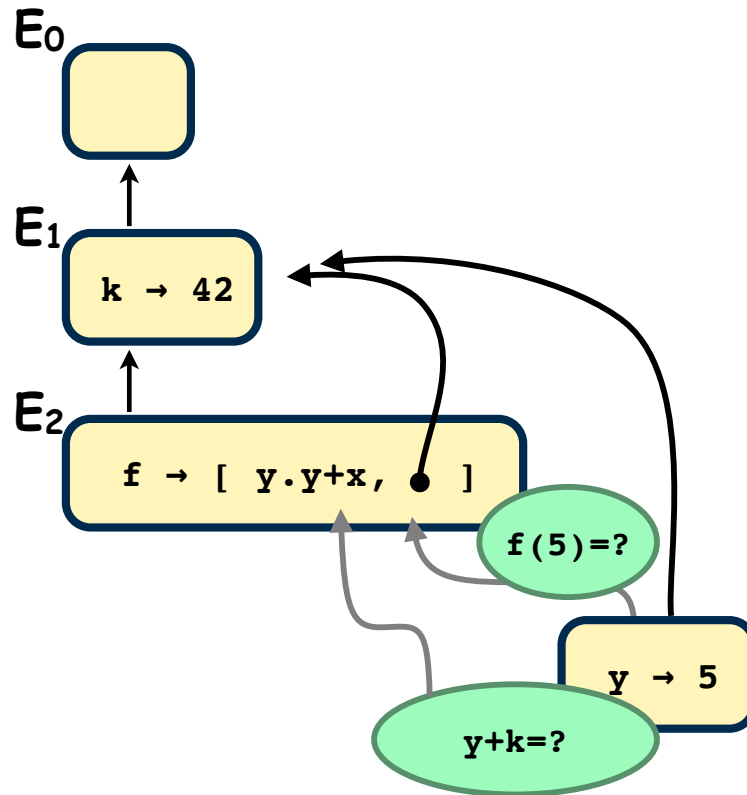
```
f(2)
```


Example Evaluation #2



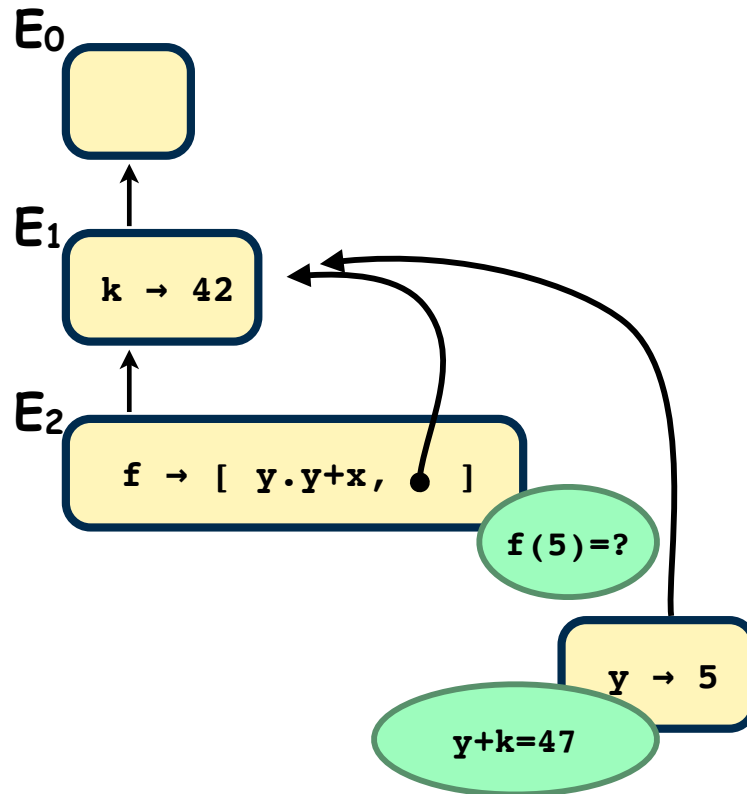
```
let k = 42 ;  
  
let f = fn y => { y+k } ;  
  
f(2)
```

Example Evaluation #2



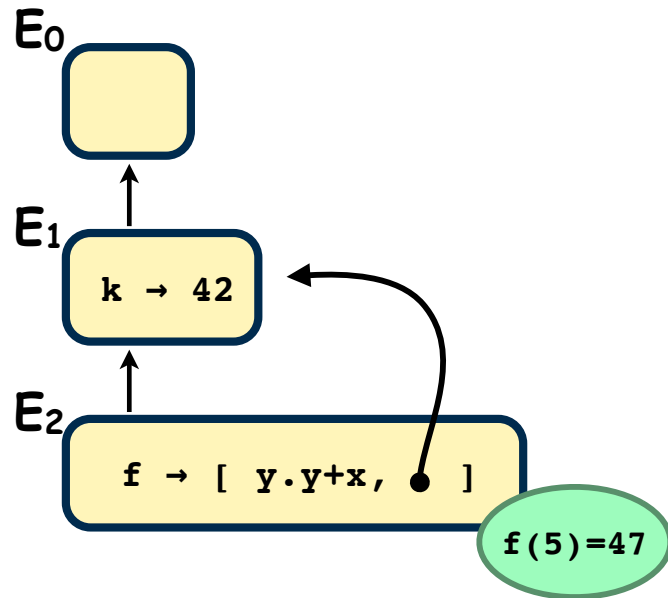
```
let k = 42 ;  
  
let f = fn y => { y+k } ;  
  
f(5)
```

Example Evaluation #2



```
let k = 42 ;  
  
let f = fn y => { y+k } ;  
  
f(5)
```

Example Evaluation #2

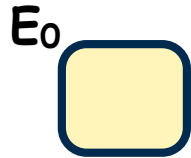


```
let k = 42 ;  
  
let f = fn y => { y+k } ;  
  
f(5)
```

Example Evaluation #3

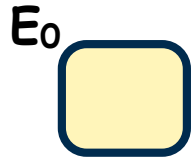
```
let x=1 ;  
  
let f = fn y -> { y+x } ;  
  
let g = fn x -> { x+f(x) } ;  
  
g(2)
```

Example Evaluation #3



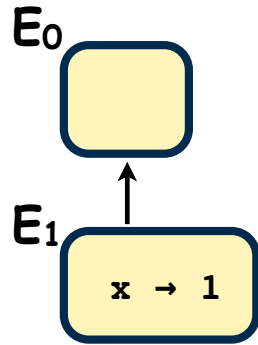
```
let x=1 ;  
  
let f = fn y -> { y+x } ;  
  
let g = fn x -> { x+f(x) } ;  
  
g(2)
```

Example Evaluation #3



```
let x=1 ;  
  
let f = fn y -> { y+x } ;  
  
let g = fn x -> { x+f(x) } ;  
  
g(2)
```

Example Evaluation #3



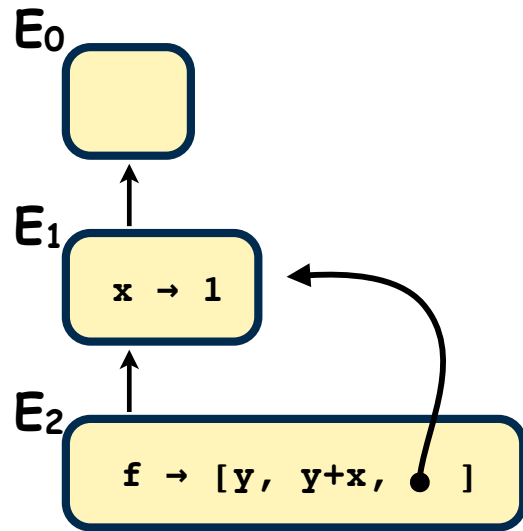
```
let x=1 ;
```

```
let f = fn y -> { y+x } ;
```

```
let g = fn x -> { x+f(x) } ;
```

```
g(2)
```


Example Evaluation #3



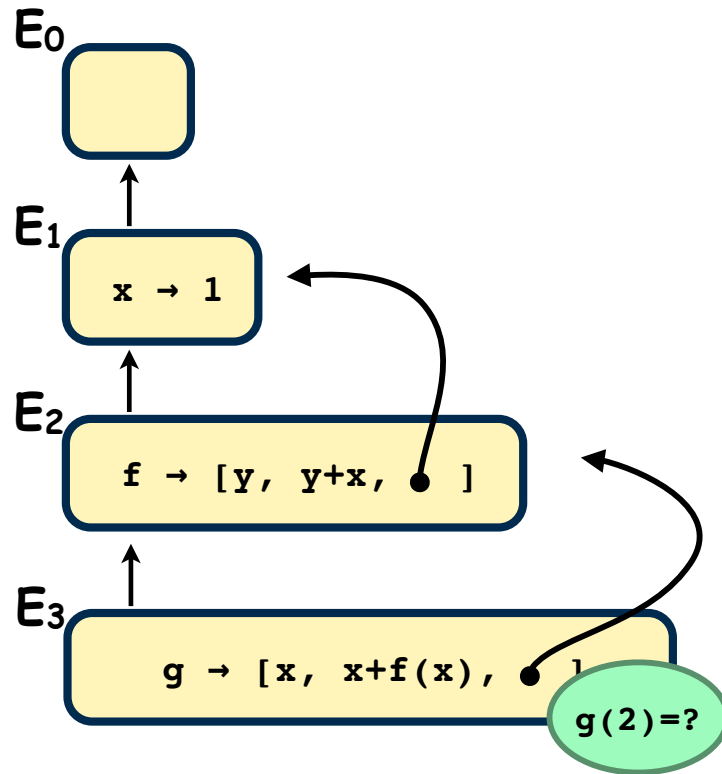
```
let x=1 ;
```

```
let f = fn y -> { y+x } ;
```

```
let g = fn x -> { x+f(x) } ;
```

```
g(2)
```

Example Evaluation #3



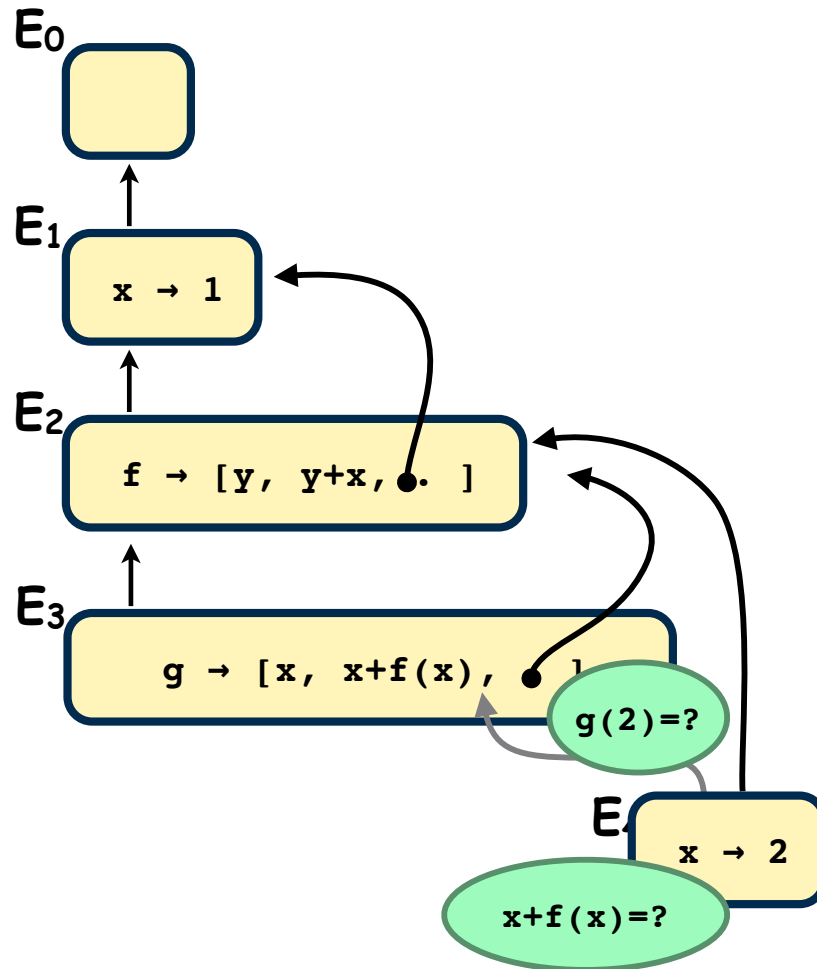
```
let x=1 ;
```

```
let f = fn y -> { y+x } ;
```

```
let g = fn x -> { x+f(x) } ;
```

```
g(2)
```

Example Evaluation #3



```
let x=1 ;
```

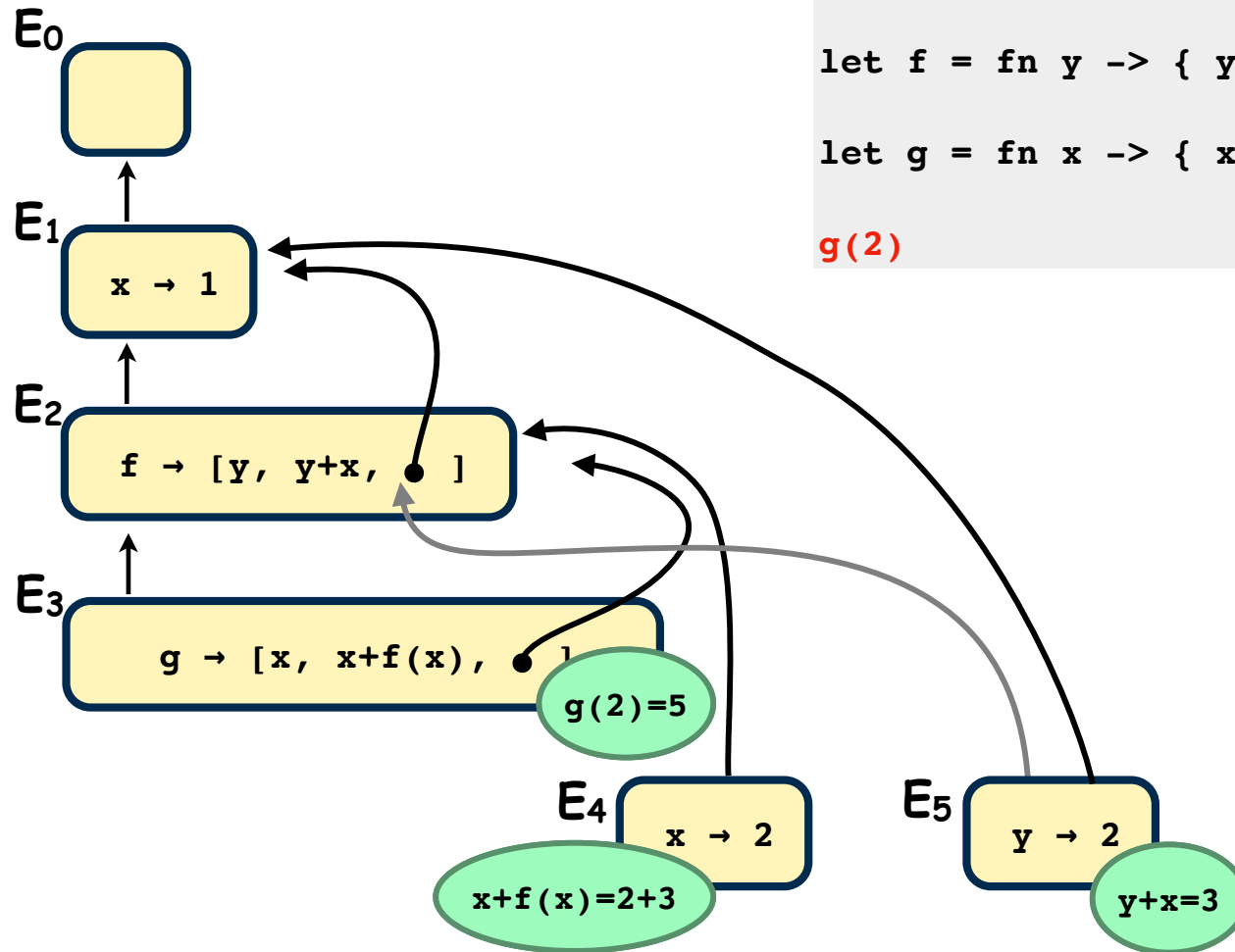
```
let f = fn y -> { y+x } ;
```

```
let g = fn x -> { x+f(x) } ;
```

```
g(2)
```

Example Evaluation #3

```
let x=1 ;  
  
let f = fn y -> { y+x } ;  
  
let g = fn x -> { x+f(x) } ;  
  
g(2)
```



M2 Sum Up

- Naming
- Binding, Bound and Free identifiers, Scope
- Safe Substitution
- Reduction Semantics
- Big Step Semantics
- Environment / Closure Semantics
- Implementation of Environments