Interpretação e Compilação de Linguagens (de Programação)

22/23

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Functional Abstraction

Every programming language needs to incorporate abstraction mechanisms.

Abstraction constructs allow parametrised expressions to be defined and used in the appropriate contexts.

We may find simple functional or procedural abstraction, used in function and procedure definitions, type abstraction, used in generics, and so on…

In this module we will study how to incorporate functional abstraction in our core programming language by introducing functional values, also called first class functions.

- functional abstraction in programming languages
- lambda abstraction
- interpretation of functional values
- closures
- typing of functional abstractions

First class functions

A first class function is a special value defined by a parametrised expression

fun $x_1, \ldots, x_n \rightarrow$ E **end**

Such expression is called an **abstraction** (or λ-abstraction)

The identifiers $x_1, \ldots, x_n$ are the abstraction **parameters**.

The parameters are binding occurrences, with scope the abstraction **body** E, where E is **any expression of the language**.

First class functions

An abstraction is a syntactical expression that denotes a function.

A function is a special value F that supports an application operation F.apply, we may apply the function F to a value v, to obtain a value as result F.apply(v)

A programming language may support functions but not abstractions (C, C++)

Most modern languages support abstraction:

Python: lambda x : x*2

Rust: |x| { x*2 }

JavaScript: (x) => x*2

OCaml: fun x -> x *2

CALCF: fun x -> x*2 end

The origin of functional abstraction is Church's lambda calculus.

The λ-Calculus

- The lambda-calculus is a minimal programming language created by Alonzo Church in 1936, it is the basis for all abstraction mechanisms used in modern programming
- It uses functions as "first class" entities, and albeit extremely simple, can represent all constructs of Turing complete programming languages.
- Syntax of the lambda-calculus:

$$\mathrm{E} ::= \quad x \quad | \quad \lambda x.\mathrm{E} \quad | \quad \mathrm{E1\ E2}$$

- Abstract syntax:

var:	string → LAMBDA	(x)
abs:	string × LAMBDA → LAMBDA	$(\lambda x.\mathrm{E})$
app:	LAMBDA × LAMBDA → LAMBDA	(E1 E2)

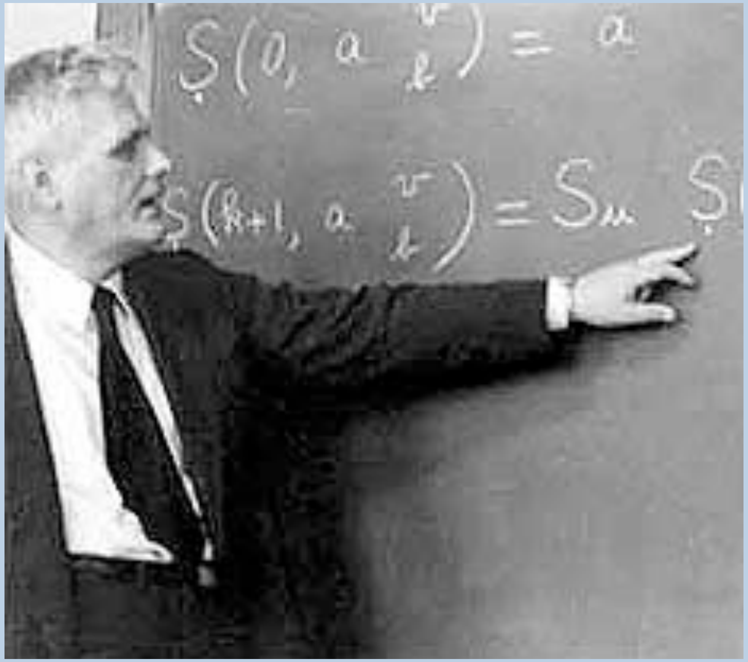
- A multi-parameter function and application is defined using "currying"

$$\lambda x_1 \lambda x_2 \ldots \lambda x_n.\mathrm{B} \qquad\qquad \mathrm{F}\ \mathrm{E}_1\ \mathrm{E}_2 \ldots \mathrm{E}_n$$

Alonzo Church (1903-1995)

Church Thesis:

“The set of all computable functions is the set of functions defined in the lambda-calculus.”



Church was Alan Turing PhD advisor.

Together, they have shown that the lambda calculus has the same computational power as the Turing machine.

The language CALCF (abstract syntax)

num:	Integer $\rightarrow$ CALCF
id:	String $\rightarrow$ CALCF
add:	CALCF $\times$ CALCF $\rightarrow$ CALCF
mul:	CALCF $\times$ CALCF $\rightarrow$ CALCF
div:	CALCF $\times$ CALCF $\rightarrow$ CALCF
sub:	CALCF $\times$ CALCF $\rightarrow$ CALCF
….	
def:	List(String $\times$ CALCF) $\times$ CALCF $\rightarrow$ CALCF
fun:	String $\times$ CALCF $\rightarrow$ CALCF
app:	CALCF $\times$ CALCF $\rightarrow$ CALCF

The language CALCF (example)

```
fun x -> x*2 end (4);;
```

The language CALCF (example)

```
def f = fun x -> x+1 end
  in
  def g = fun y -> f(y)+2 end
    in
    def x = g(2)
      in
       x+x
      end
  end
end;;
```

Semantics of CALCF

- Algorithm eval( ) that computes the value of any **open** CALCF expression:

$$\text{eval} : \text{CALCF} \times \text{ENV}\langle\text{Value}\rangle \rightarrow \text{Value}$$

CALCF = open programs

ENV = environments

Value = Bool U Integer U Closure U {ERROR}

Semantics of CALCF

- Algorithm eval( ) that computes the value of any **open** CALCF expression:

$$\text{eval} : \text{CALCF} \times \text{ENV}\langle\text{Value}\rangle \rightarrow \text{Value}$$

CALCF = open programs

ENV = environments

Value = Bool U Integer U Closure U {ERROR}

Closures are special values used to represent functions.

A closure is a triple of the form [id, E, *env*]

id is an identifier (the closure parameter)

E is an expression (the closure body)

env is an environment (must declare all free names in body except id)

- The closure body E may be evaluated by providing some value for the parameter id (this will correspond to function call)

CALCF Interpreter

- Algorithm eval( ) that computes the value of any **open** CALCF expression:

$$\mathrm{eval} : \mathrm{CALCI} \times \mathrm{ENV}\langle\mathrm{Value}\rangle \rightarrow \mathrm{Value}$$

eval(**num**(*n*) , *env*) ≜ *n*

eval(**id**(s) , *env*) ≜ *env*.Find(s)

eval(**add**(E1,E2) , *env*) ≜ **eval**(E1, *env*) + **eval**(E2, *env*)

eval(**abs**(s, E), *env*) ≜ **{** return **[** s, E, *env* **]** ; }

```
eval( app(E1, E2), env) ≜ { v1 = eval(E1,env); v2 = eval(E2,env);
                             if v1 is [ param, B, env0 ] then
                                 e = env0. beginScope();
                                 e.assoc(param,v2);
                                 result = B.eval(e);
                                 e = e. endScope();
                                 return result;
                            else ERROR
                            }
```

Example Evaluation

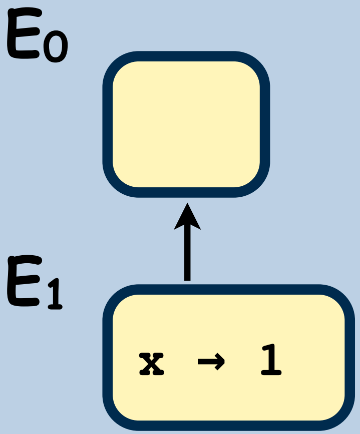
```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
    end
  end
end
```

Example Evaluation

E0

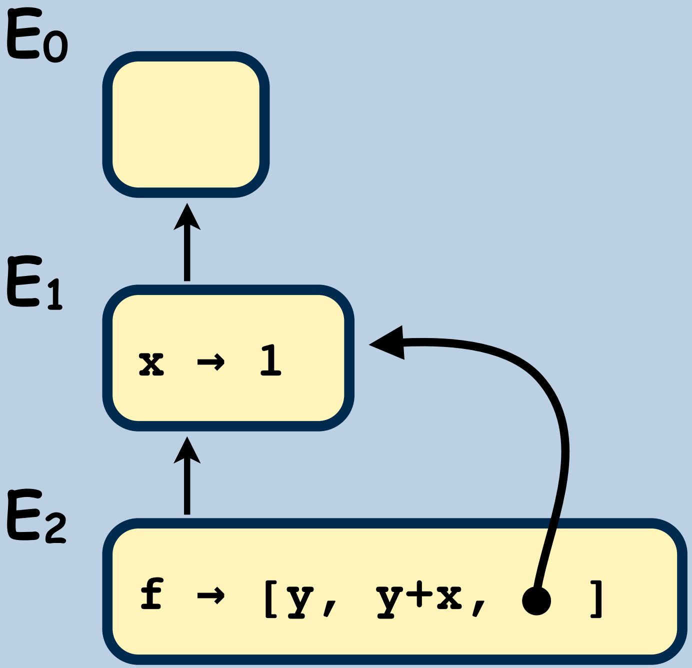
```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2) end end end;;
```

Example Evaluation



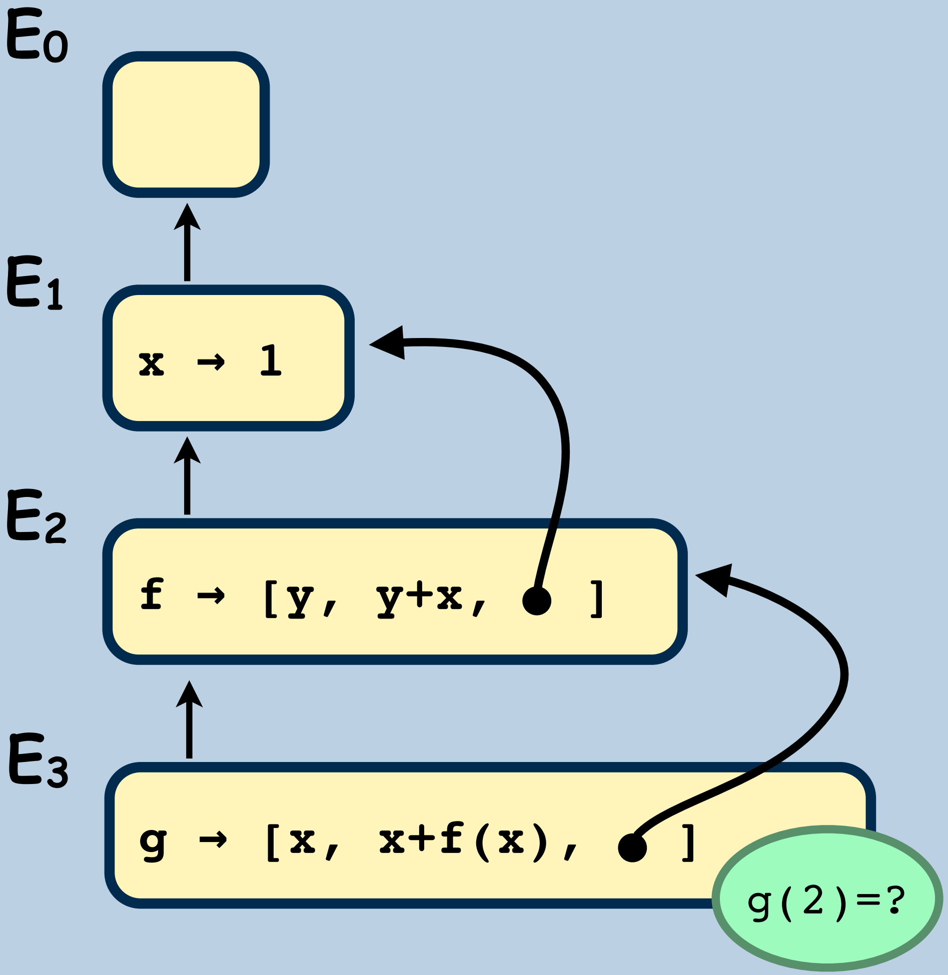

```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

Example Evaluation



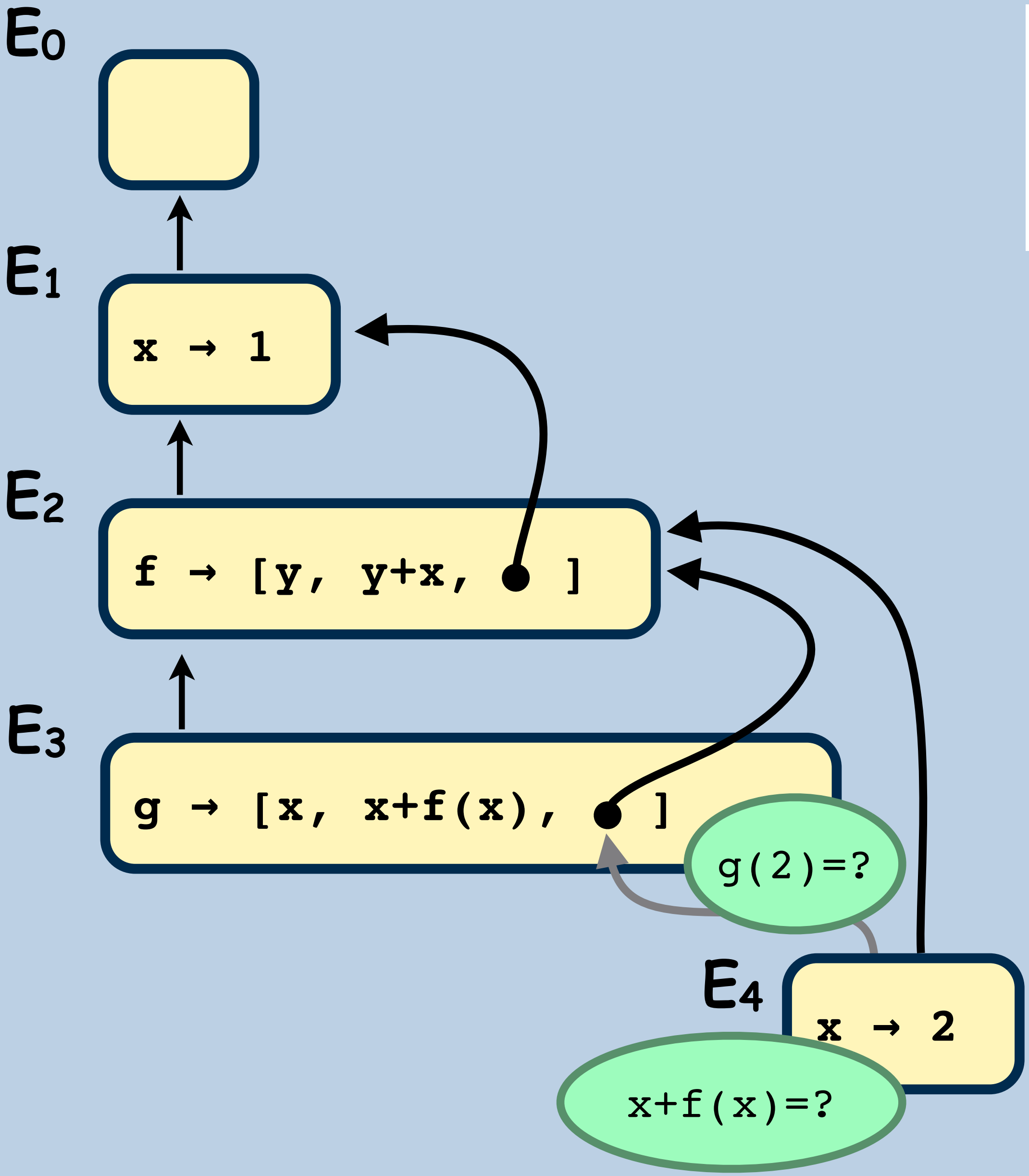

```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

Example Evaluation




```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

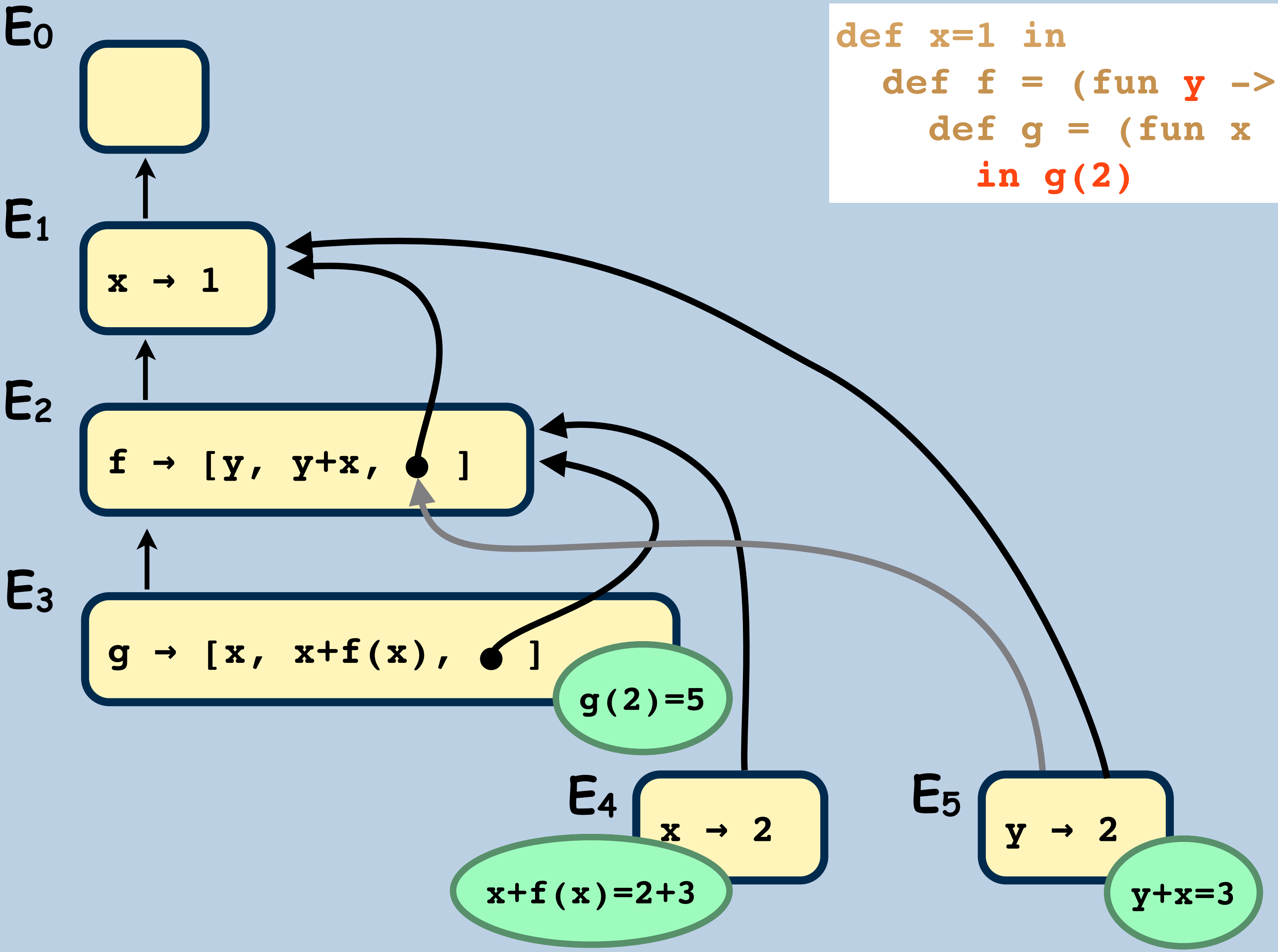
Example Evaluation




```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

Example Evaluation

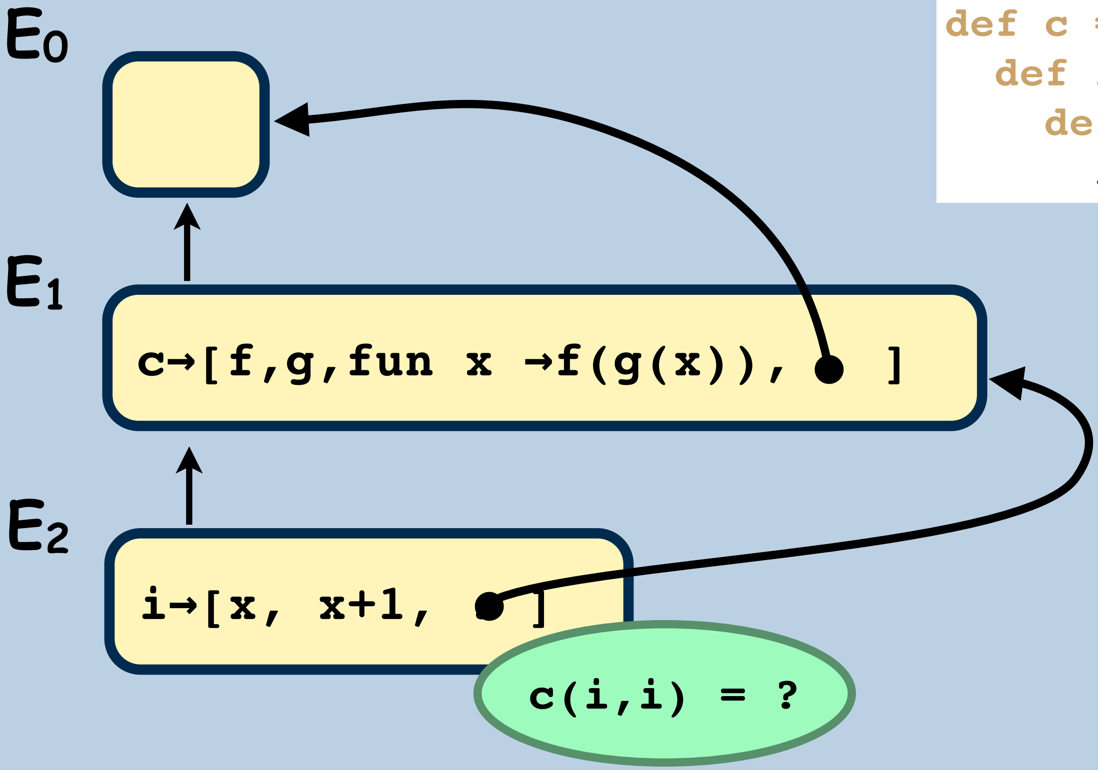
```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```



Example Evaluation

```
def comp = (fun f,g -> (fun x -> f(g(x))))
  in
  def inc = (fun x -> x+1)
    in
    def dup = comp(inc,inc)
      in dup(2)
    end
  end
end
```

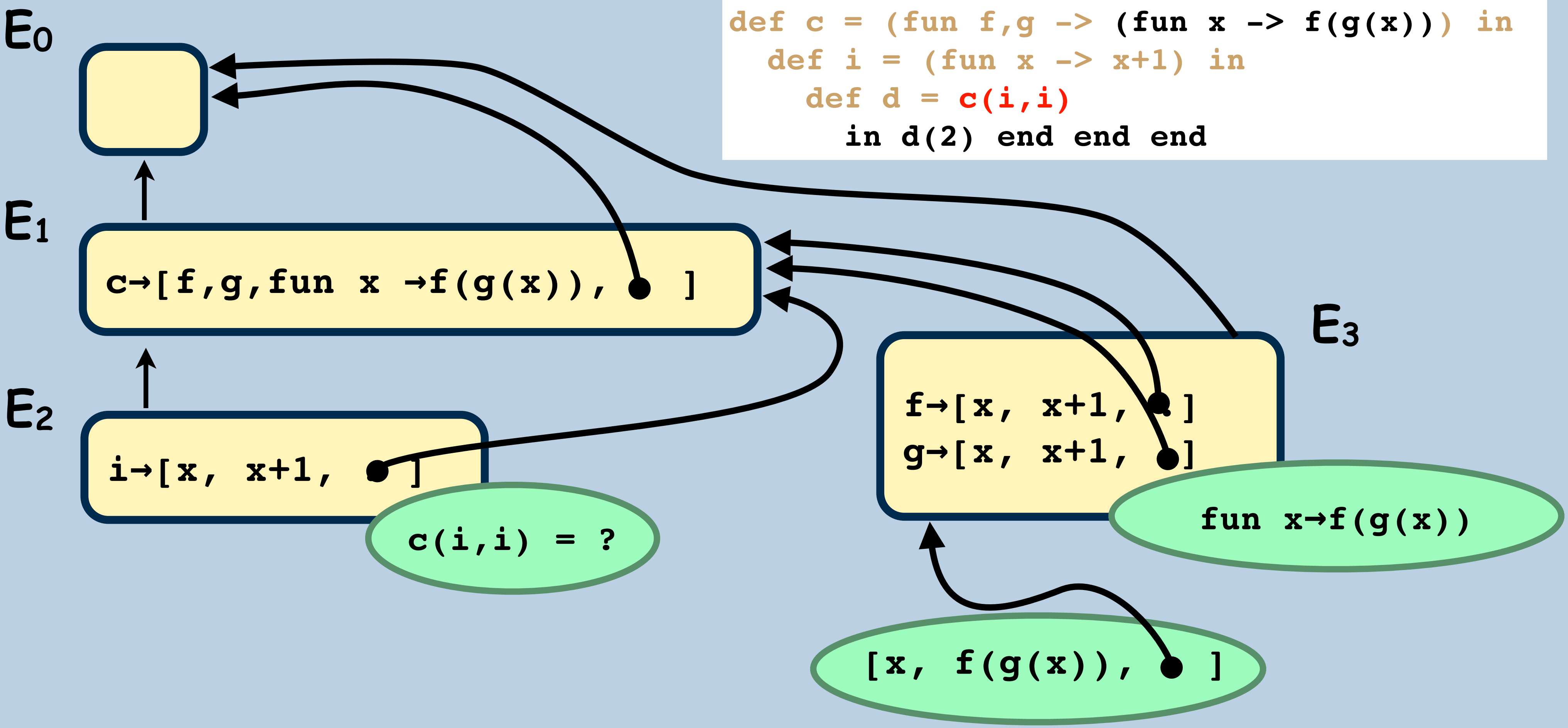
Example Evaluation




```
def c = (fun f,g -> (fun x -> f(g(x))) in
  def i = (fun x -> x+1) in
    def d = c(i,i)
      in d(2) end end end
```

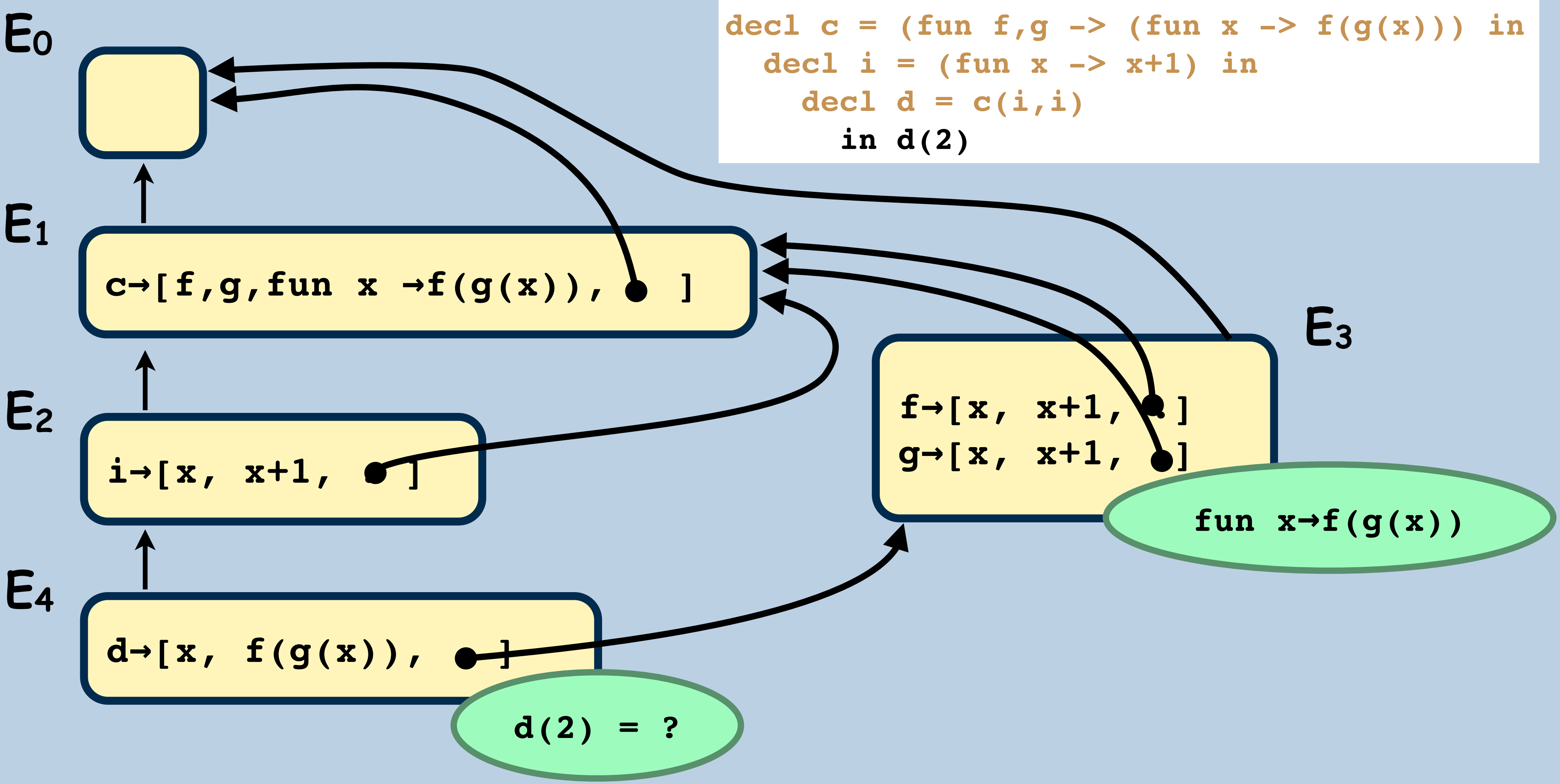
Example Evaluation

```
def c = (fun f,g -> (fun x -> f(g(x))) in
  def i = (fun x -> x+1) in
    def d = c(i,i)
      in d(2) end end end
```



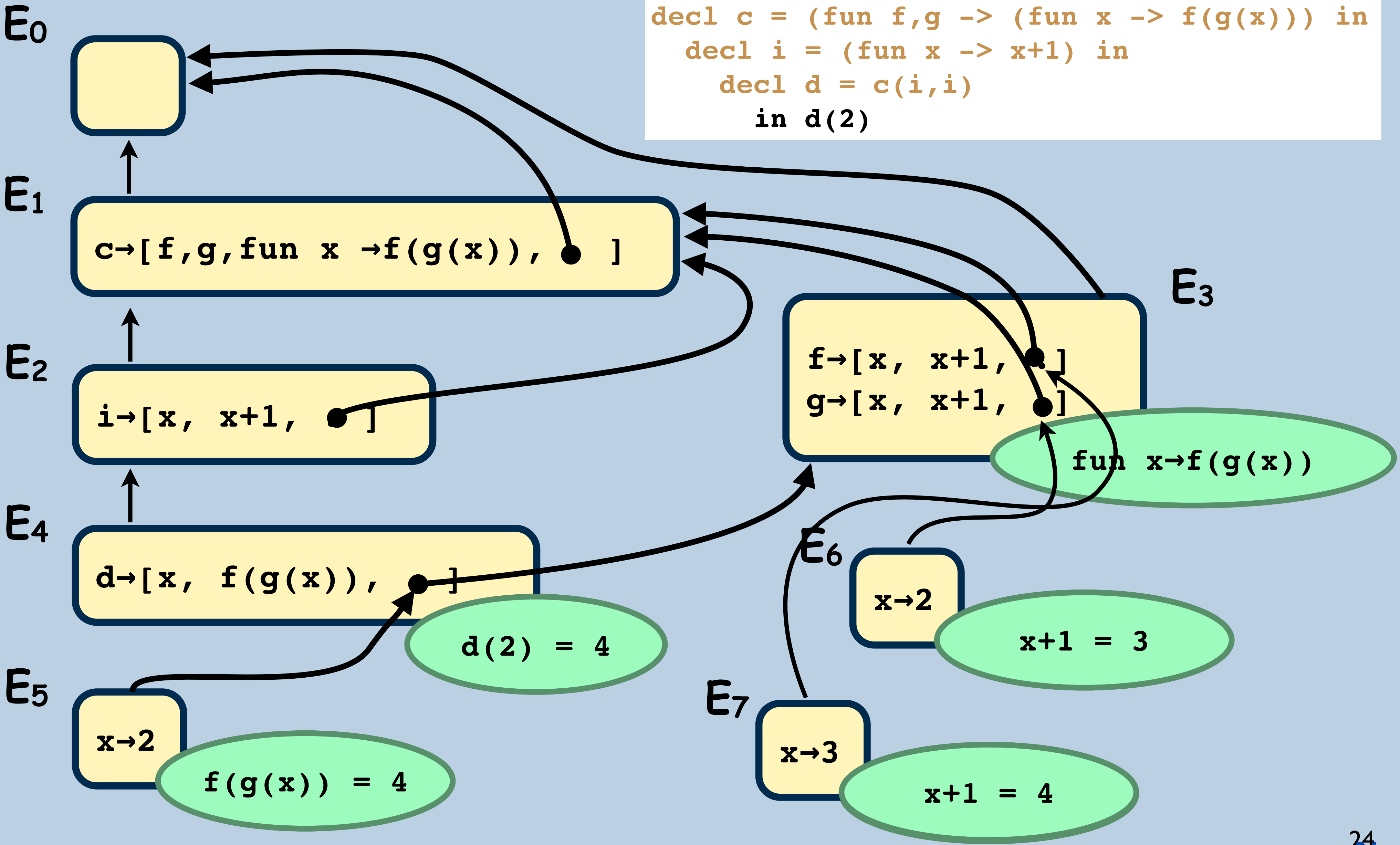
Example Evaluation

```
decl c = (fun f,g -> (fun x -> f(g(x))) in
  decl i = (fun x -> x+1) in
    decl d = c(i,i)
      in d(2)
```



Example Evaluation

```
decl c = (fun f,g -> (fun x -> f(g(x))) in
  decl i = (fun x -> x+1) in
    decl d = c(i,i)
      in d(2)
```



CALCF Types and Typechecking

- Map **eval** to compute a value + effect for any CALCF programs:

$$\text{eval} : \text{CALCS} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

- Map **typchk** that computes a type for a CALCF program:

$$\text{typechk} : \text{CALCS} \times \text{ENV} \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \ \text{bool}, \ \text{ref}\{\text{TYPE}\}, \text{fun}[\text{TYPE},\text{TYPE}] \}$$

CALCF Types and Typechecking

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCF × ENV<TYPE> → TYPE U { TYPEERROR }

  ENV : ID → TYPE

  TYPE = { int, bool, ref{TYPE},fun{TYPE,TYPE} }

int: type of integer values.

bool: type of boolean values.

ref{$\mathcal{T}$}: type of references that may only hold values of type $\mathcal{T}$.

fun{$\mathcal{A}$,$\mathcal{B}$}: type of functions that take arg of type $\mathcal{A}$ and return a value of type $\mathcal{B}$

Example: fun{int,bool} is the type functions that take an integer as argument and return a boolean

CALCF Typing Rules

- Map **typchk** that computes a type for any CALCF program:

  typechk : CALCF × ENV<TYPE> → TYPE U { TYPEERROR }

```
typchk( fun(s:T0, E) , env ) ≜ {
    env0 = env.beginScope();
    env 0= env0.assoc(s,T0);
    t1 =  typchk ( E, env0);
    env.endScope();
    if (t1 == TYPEERROR) then return t1
    else return fun(T0,t1);
    }
```

CALCF Typing Rules

- Map **typchk** that computes a type for any CALCF program:

  typechk : CALCF × ENV<TYPE> → TYPE U { TYPEERROR }

```
typchk( fun(s:T0, E) , env ) ≜ {
        env0 = env.beginScope();
        env 0= env0.assoc(s,T0);
        t1 =  typchk ( E, env0);
        env.endScope();
        if (t1 == TYPEERROR) then return t1
        else return fun(T0,t1);
        }
```

we need to declare the argument type!

CALCF Typing Rules

- Untyped functional abstraction gives rise to polymorphic functions. Consider the example:

```
fun x -> new x end
```

- The most general type for this function is fun[X,ref X], for all types X. This could be expressed by a **polymorphic type** scheme [Damas-Milner82] of the form All X. fun[X,ref X] type, but we don't want to get into such advanced topic in this course.

- So, in our language, we will annotate function arguments with types, e.g.,

```
fun x:int => new x end
```

```
fun f:(int)int -> f (3) end :: ((int)int)int
```

CALCF Typing Rules

- Map **typchk** that computes a type for any CALCF program:

  typechk : CALCF × ENV<TYPE> → TYPE U { TYPEERROR }

```
typchk( app(E1, E2) , env ) ≜ { t1 =  typchk ( E1, env);
                                 if (t1 is fun(T0,TR)) then
                                     t2 =  typchk ( E2, env);
                                     if (t2 != T0) then return TYPEERROR;
                                     return TR;
                                 else return TYPEERROR
                                 }
```

Sample Typed Program

```
def
   mod : (int, int)int = fun dividend:int, divisor:int ->
      dividend - divisor * (dividend / divisor)
   end

   seed : ref int = new 2

   random : ()int =
      fun ->   seed := mod(8121 * !seed + 28411, 181); !seed
   end

   is_inside_circle : (int, int)bool =
      fun x:int, y:int -> (x * x) + (y * y) <= 32767
   end
in
   def
      i : ref int = new 0
      s : ref int = new 0
   in
      while !i < 30000 do
         def
            x : int = random()
            y : int = random()
         in
            if is_inside_circle(x, y) then  s := !s + 1
            else  !s
            end;
            i := !i + 1
         end
      end;
      println 4 * !s * 100 / 30000
   end
end;;
```

Sample Typed Program

```
def  glo = new 0
  in def f  = fun n:int -> glo := !glo + n end
  in
    f(2);
    f(3);
    f(4);
    println !glo
end;;
```

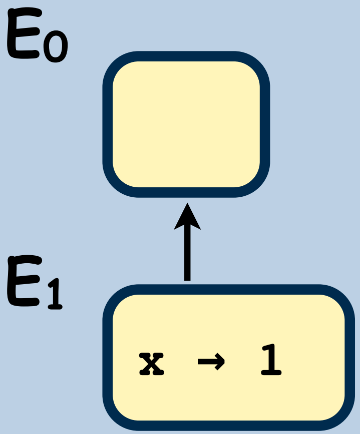
CALCF Compilation

Example Evaluation

E_0

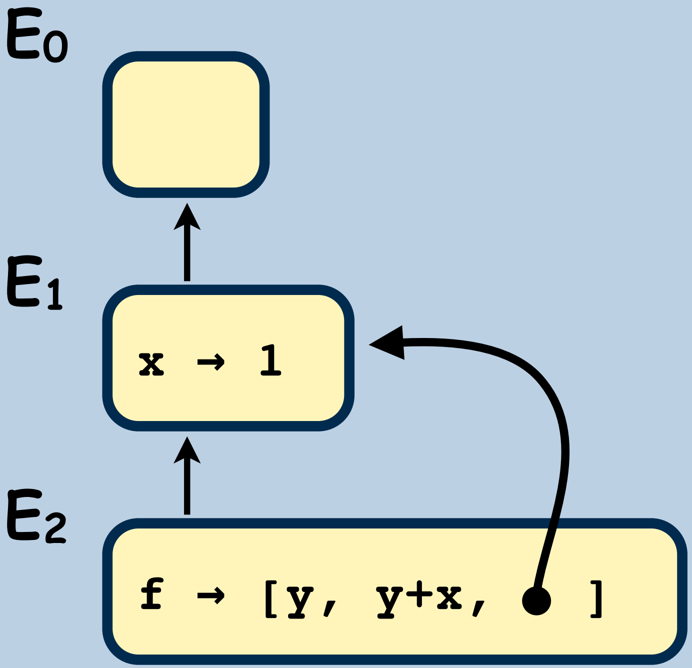
```
def x=1 in
  def f = (fun y -> x+y) in
     in f(2) end ;;
```

Example Evaluation



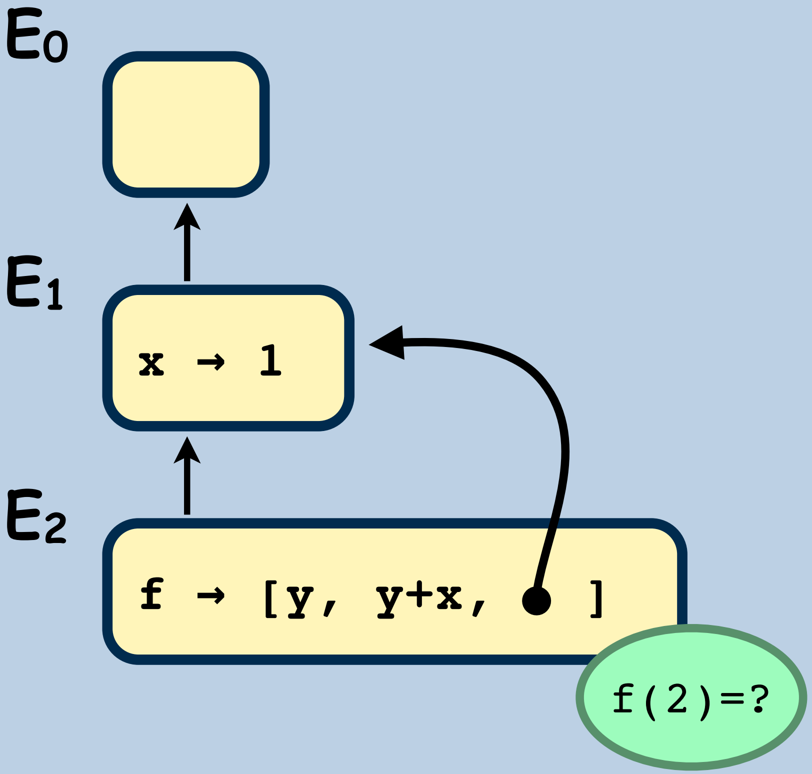

```
def x=1 in
  def f = (fun y -> x+y) in
     in f(2) end ;;
```

Example Evaluation



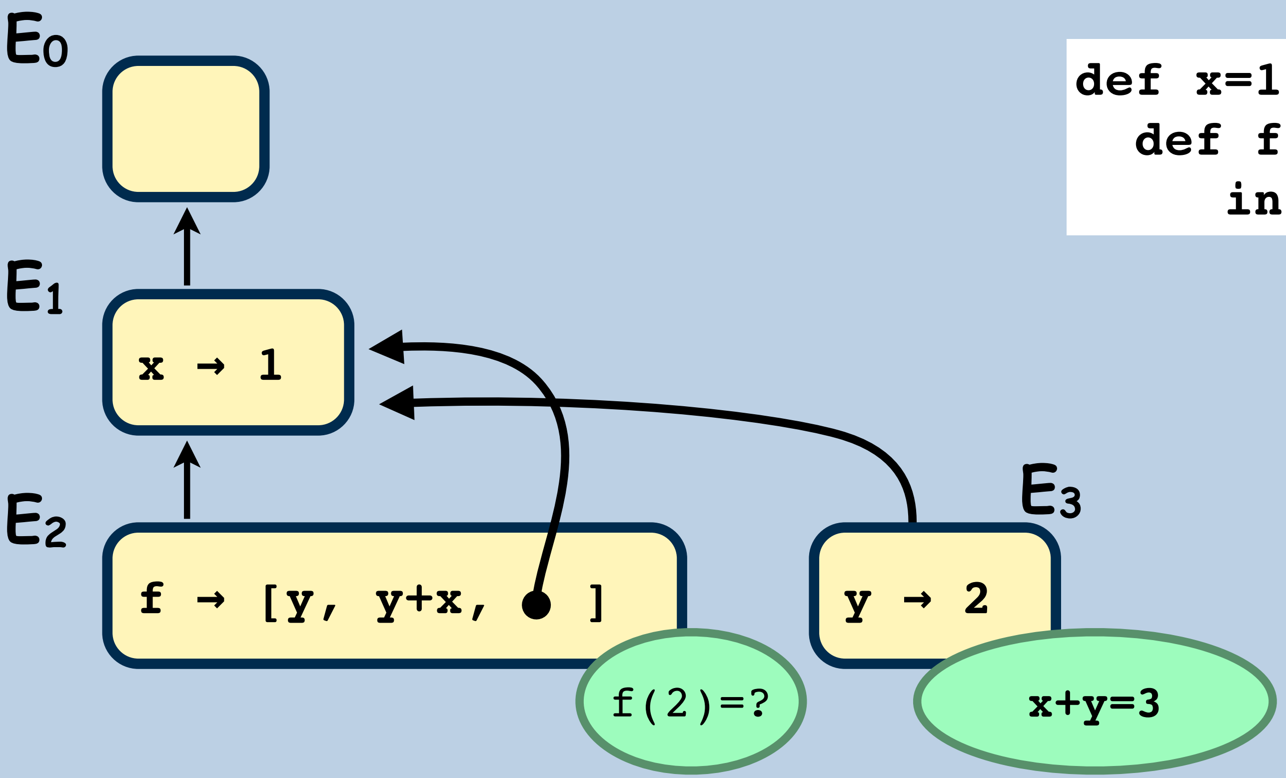

```
def x=1 in
  def f = (fun y -> x+y) in
     in f(2) end ;;
```

Example Evaluation



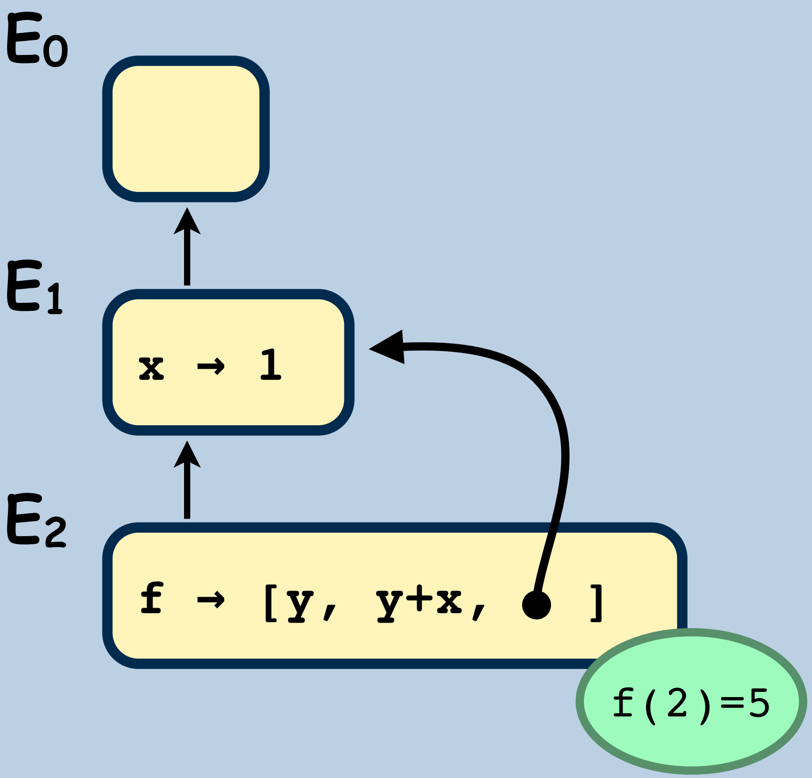

```
def x=1 in
  def f = (fun y -> x+y) in
     in f(2) end ;;
```

Example Evaluation




```
def x=1 in
  def f = (fun y -> x+y) in
     in f(2) end ;;
```

Example Evaluation




```
def x=1 in
  def f = (fun y -> x+y) in
     in f(2) end ;;
```

CALCF Compiler (environment based)

- Algorithm compile( ) that generates machine code for any well-typed **open** CALCF expression:

$$\text{eval} : \text{CALCF} \times \text{ENV<Coord>} \rightarrow \text{CodeSeq}$$

CALCF Compiler (environment based)

- Functions are represented as closures at runtime
- Closures are represented by JVM objects F which implement an apply operation (JVM method)
- F.apply(v) returns the result if applying closure F to value v

CALCF Compiler (environment based)

- For each closure created by the program (while compiling a lambda abstraction fun x -> E end) the compiler needs to generate an specific closure JVM class
- The closure object contains an apply method, whose body is composed essentially by the code [[E]] for the function body E plus some environment manipulation operations.
- The function call will be essential be analogous to the "equivalent" application to declaration expansion

fun x1,...,xk -> E end (E1, .. Ek)

≈

decl x1 = E1 ... xk = Ek in E

Compilation of function abstraction

[[fun x1,…,xm -> E end]]D =

```
new closure_id
dup
aload SL
putfield closure_id/SL LSLtype;
```

closure_id is the gensym generated code for a new closure class (e.g., clos_0002).

Such closure object has a field used to hold the closure environment, and an apply method to be called whenever the closure needs to be applied to value arguments.

Compilation of function call

[[E (E1,…,Ek)]]D =

```
[[ e ]]D
checkcast closure_interface_id
[[ e1 ]]D
....
[[ ek ]]D
invokeinterface closure_interface_id/apply(type1;type2;...)type
```

closure_interface_id is a generated interface name for the type of the new closure class (e.g., ftype_0002).

Such interface defines the type of the apply method (from typechecking)

Note that function call is based on the type of the function not on the actual function code, which is unknown at call site.

JVM instructions

invokeinterface

Operation

Invoke interface method

Format

```
invokeinterface
indexbyte1
indexbyte2
count
0
```

Forms

invokeinterface = 185 (0xb9)

Operand Stack

..., *objectref*, [*arg1*, [*arg2* ...]] →

...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to an interface method (§5.1), which gives the name and descriptor (§4.3.3) of the interface method as well as a symbolic reference to the interface in which the interface method is to be found. The named interface method is resolved (§5.4.3.4).

Generation of closure class

[[fun x1,…,xk -> E end]]D =

```
.class closure_id
.implements ftype_id
.field public SL Lframe_id;
method apply(type1;type2;…)returntype
// here code for method apply
.end method
```

Compilation scheme of closure class

```
method apply(type1;type2;…)rtype
,limits locals k+2
new fid_frame  // create function stack frame (aka activation record)
dup
aload 0 // get reference to "this"
getfield closure_id/SL LSLtype  // get the env stored in this closure
putfield fid_frame/sl LSLtype    // link to activation record
dup
aload 1   // load first function argument
putfield fid_frame/x1 x1argtype // store it in slot x1
dup
....
aload k // load kth function argument
putfield fid_frame/xk xkargtype // all argument values are now stored in frame
astore SL // note ! this sl local is now in the method
[[ e ]]D+{x1->coordx1, … xk->coordxk}
return
```

Generation of closure activation record

```
class fid_frame
.super java/lang/Object
.field public sl Lancestor_frame_id;
.field public x0 arg1type
.field public x1 arg1type;
..
.field public xk argntype;
```

This is completely analogous to the frame class for the "equivalent" application to declaration expansion

fun x1,...,xk -> E end (E1, .. Ek)

~~

decl x1 = E1 ... xk = Ek in E

Example (code generated for function)

```
def
   inc : (int)int = fun x:int -> x + 1 end
in
     println inc(2+2)
end;;

.class public closure_0
.super java/lang/Object
.implements closure_interface_int_int
.field public sl Ljava/lang/Object;
.method public <init>()V
        aload_0
        invokenonvirtual java/lang/Object/<init>()V
        return
.end method

.class public frame_1
.super java/lang/Object
.field public sl Ljava/lang/Object;
.field public v0 I
.method public <init>()V
      aload_0
      invokenonvirtual java/lang/Object/<init>()V
      return
.end method
```

```
.interface public closure_interface_int_int
.super java/lang/Object
.method public abstract apply(I)I
.end method


.method public apply(I)I
        .limit locals 4
        .limit stack 256
        new frame_1
        dup
        invokespecial frame_1/<init>()V
        dup
        aload_0
        getfield closure_0/sl Lframe_0;
        putfield frame_1/sl Lframe_0;
        dup
        iload 1
        putfield frame_1/v0 I
        astore_3
        aload_3
        getfield frame_1/v0 I
        sipush 1
        iadd
        ireturn
.end method
```

Example (translation of source function type into JVM interface type)

```
def
   inc : (int)int = fun x:int -> x + 1 end
in
     println inc(2+2)
end;;

.class public closure_0
.super java/lang/Object
.implements closure_interface_int_int
.field public sl Ljava/lang/Object;
.method public <init>()V
        aload_0
        invokenonvirtual java/lang/Object/<init>()V
        return
.end method

.class public frame_1
.super java/lang/Object
.field public sl Ljava/lang/Object;
.field public v0 I
.method public <init>()V
      aload_0
      invokenonvirtual java/lang/Object/<init>()V
      return
.end method
```

```
.interface public closure_interface_int_int
.super java/lang/Object
.method public abstract apply(I)I
.end method


.method public apply(I)I
        .limit locals 4
        .limit stack 256
        new frame_1
        dup
        invokespecial frame_1/<init>()V
        dup
        aload_0
        getfield closure_0/sl Lframe_0;
        putfield frame_1/sl Lframe_0;
        dup
        iload 1
        putfield frame_1/v0 I
        astore_3
        aload_3
        getfield frame_1/v0 I
        sipush 1
        iadd
        ireturn
.end method
```

Translation of source types into JVM types

[[int]] = I

[[boolean]] = Z

[[ref T]] = ref_[[T]]

where this class will implement
.method public set(L[[T]];)V
.method public set()L[[T]]

[[(T1,...Tn)R]] = closure_intf_[[T]]

where this interface will implement
.method public abstract apply(L[[T1]] ;...[[Tn]];)[[R]]

Example (generation of closure class and its activation frame)

```
def
   inc : (int)int = fun x:int -> x + 1 end
in
     println inc(2+2)
end;;

.class public closure_0
.super java/lang/Object
.implements closure_interface_(I)I
.field public sl Ljava/lang/Object;
method public apply(I)I
… // see on the right
.end method
.method public <init>()V
        aload_0
        invokenonvirtual java/lang/Object/<init>()V
        return
.end method

.class public frame_1
.super java/lang/Object
.field public sl Ljava/lang/Object;
.field public v0 I
.method public <init>()V
     aload_0
     invokenonvirtual java/lang/Object/<init>()V
     return
```

```
.interface public closure_interface_(I)I
.super java/lang/Object
.method public abstract apply(I)I
.end method


.method public apply(I)I
        .limit locals 4
        .limit stack 256
        new frame_1
        dup
        invokespecial frame_1/<init>()V
        dup
        aload_0
        getfield closure_0/sl Lframe_0;
        putfield frame_1/sl Lframe_0;
        dup
        iload 1
        putfield frame_1/v0 I
        astore_3
        aload_3
        getfield frame_1/v0 I
        sipush 1
        iadd
        ireturn
.end method
```

Recursion and Recursive Definitions

Recursion and Recursive Definitions

- Our basic language forbids recursive definitions
- They do not make sense in general

```
def x = 2 in
 (def x = x+2    // here x in x+2 refers to the previous binding x = 2
    in
       x + x
  end) + x
end
```

Recursion and Recursive Definitions

- Our basic language forbids recursive definitions
- What if we allow definitions of a name to refer to the itself ?

```
(def x = x+2    // here x in x+2 refers to the current binding x = x+2
   in
     x + x
 end) + x
```

Recursion and Recursive Definitions

- Our basic language forbids recursive definitions
- What if we allow definitions of a name to refer to the itself ?

```
(def x = 2
   in def
        f = fun y -> …g … x end
        g = fun z ->   ..f …
        x = 2
     in
 end) + x
```

Recursion and Recursive Definitions

- Our basic language forbids recursive definitions
- Obviously this does not make sense, there is no x such that x = x + 2!

```
(def x = x+2    // here x in x+2 refers to the current binding x = x+2
   in
     x + x
 end) + x
```

Recursion and Recursive Definitions

- Our basic language forbids recursive definitions
- The initialisation expression E needs to be evaluated before a value for the binding of x is know, if the name x already occurs in E we cannot compute it, clearly there is a circularity

```
(def x = x+2    // here x in x+2 refers to the current binding x = x+2
   in
     x + x
 end) + x
```

Recursion and Recursive Definitions

- Our basic language forbids recursive definitions
- However, certain kinds of "circular" or recursive definitions are possible, if the name being defined is a function or a structure.

```
(def f = fun x -> if x = 0 then 1 else x*f(x-1) end
 // here f in fun x -> …  f(x-1) end refers to the current binding f =
  in
     f (10)
 end
```

Recursion and Recursive Definitions

- Our basic language forbids recursive definitions
- The value of f is not needed when the initialisation expression. It occurs in the body of the function, that is not evaluated now, only when the closure is applied (say in the call f(10)).
- So the evaluation of is not problematic, provided the environment stored in the closure contains the proper binding for f

```
(def f = fun x -> if x = 0 then 1 else x*f(x-1) end
 // here f in fun x -> …  f(x-1) end refers to the current binding f =
  in
      f (10)
 end)
```

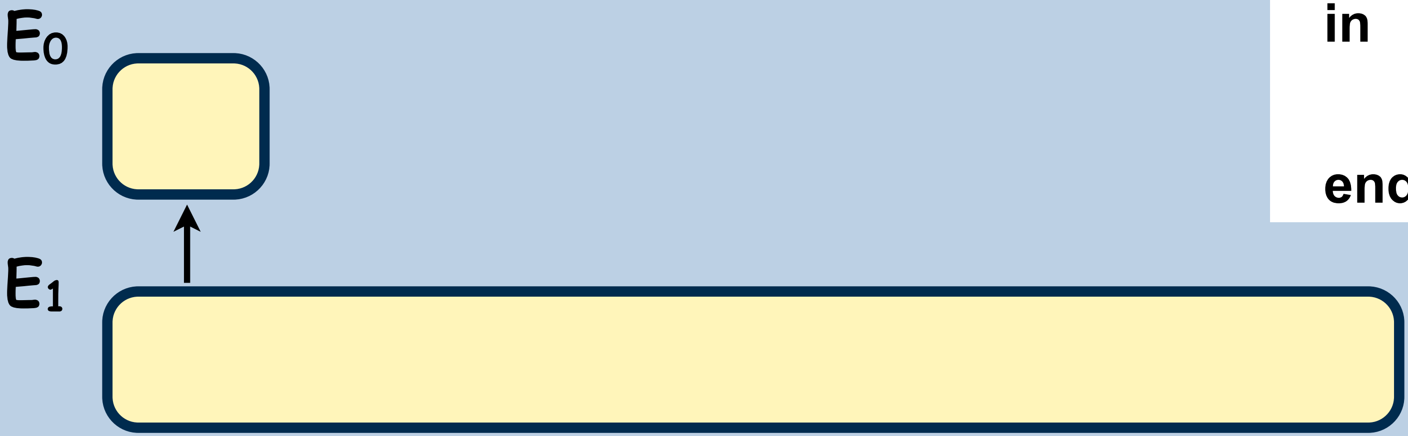
Example Evaluation

```
(def f =
  fun x -> if x = 0 then 1
               else x*f(x-1) end
  in
     f (2)
  end)
```

E_0

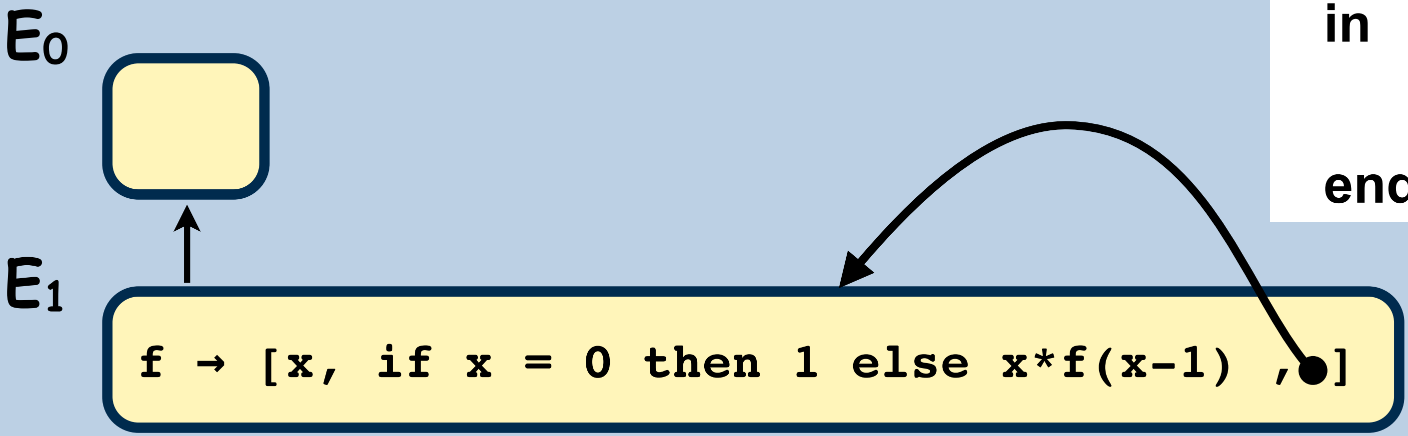
Example Evaluation

```
(def f =
  fun x -> if x = 0 then 1
              else x*f(x-1) end
  in
     f (2)
  end)
```



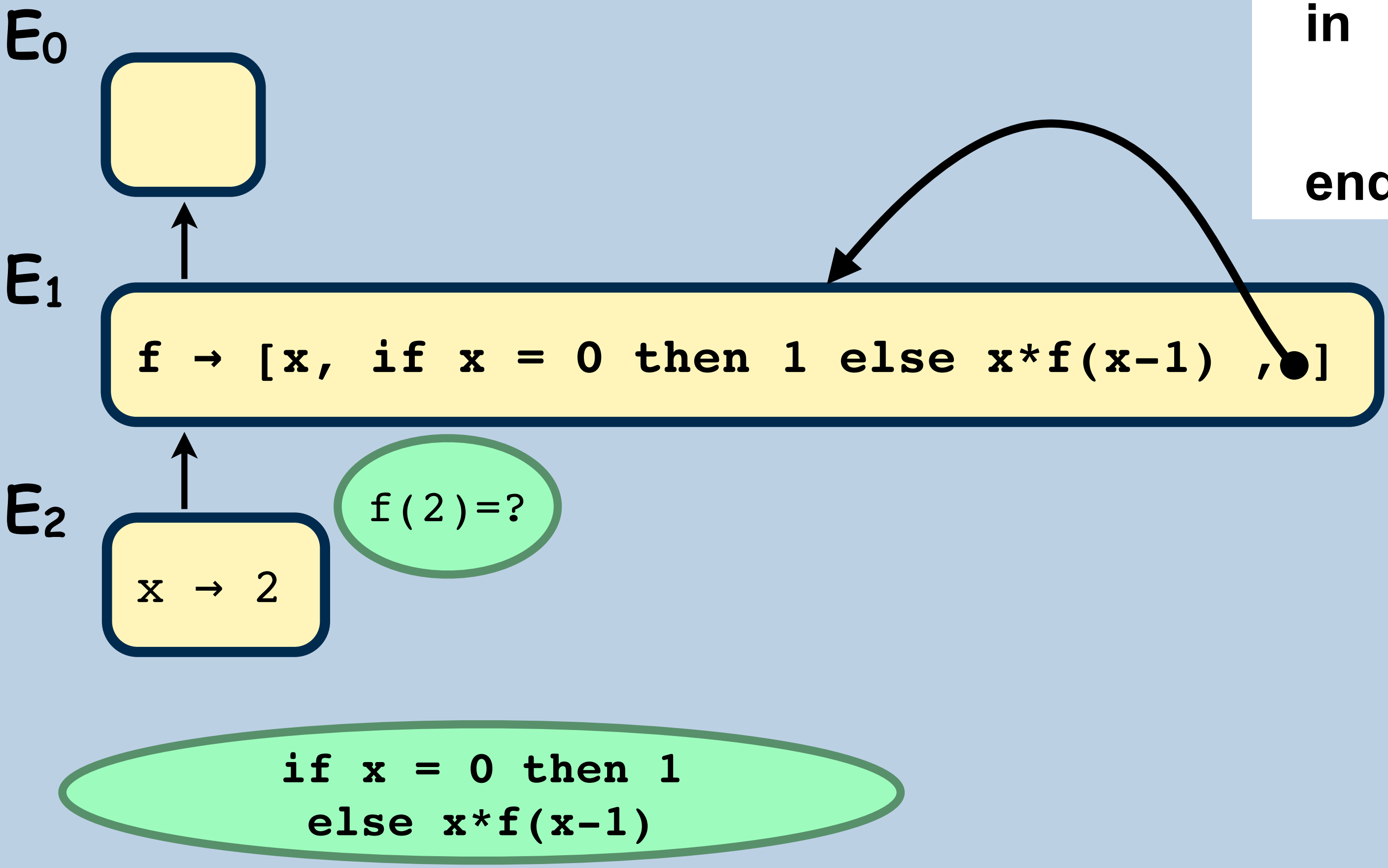
Example Evaluation

```
(def f =
  fun x -> if x = 0 then 1
              else x*f(x-1) end
  in
     f (2)
  end)
```



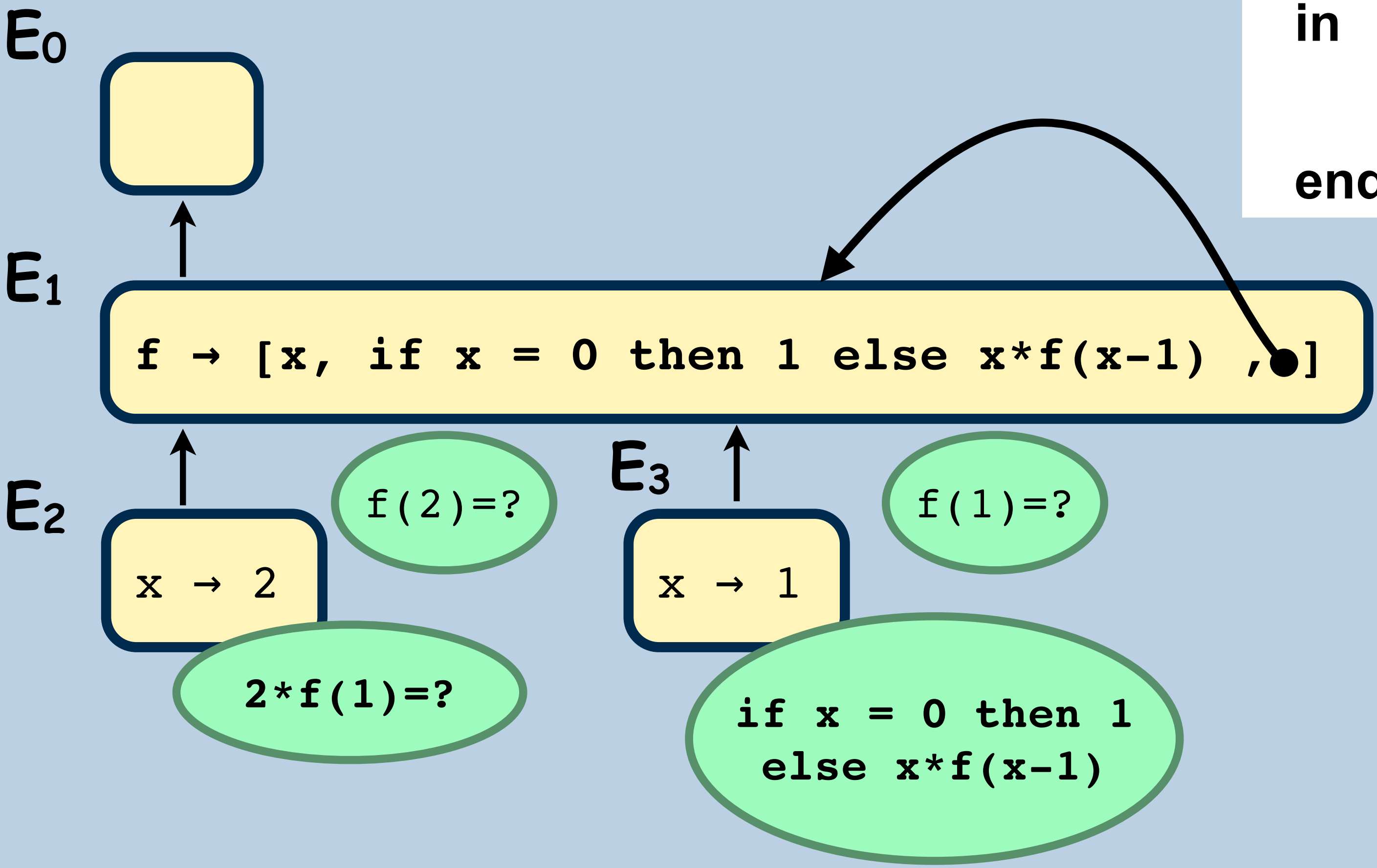
Example Evaluation

```
(def f =
  fun x -> if x = 0 then 1
                else x*f(x-1) end
  in
      f (2)
  end)
```



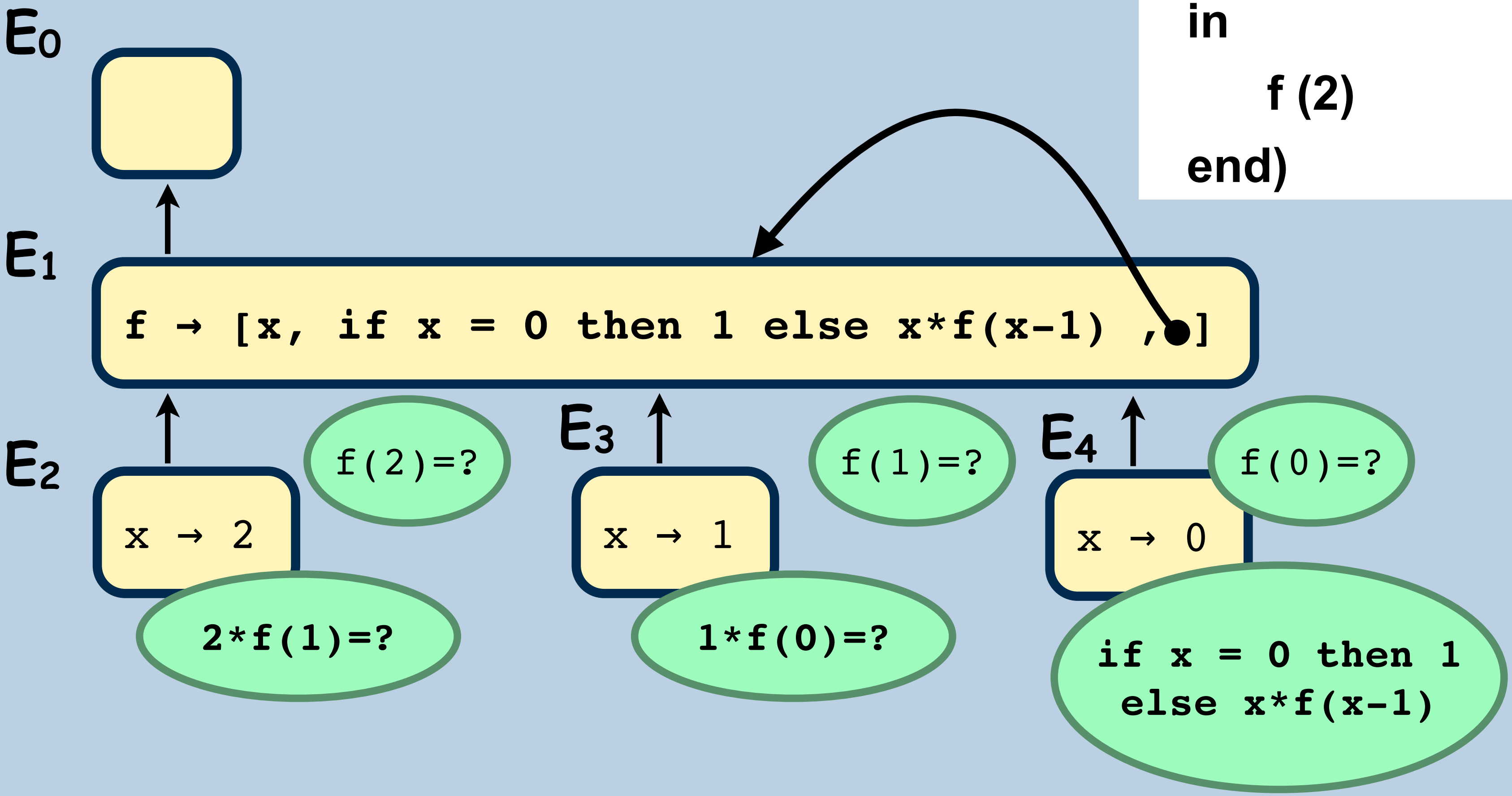
Example Evaluation

```
(def f =
  fun x -> if x = 0 then 1
                 else x*f(x-1) end
  in
       f (2)
  end)
```



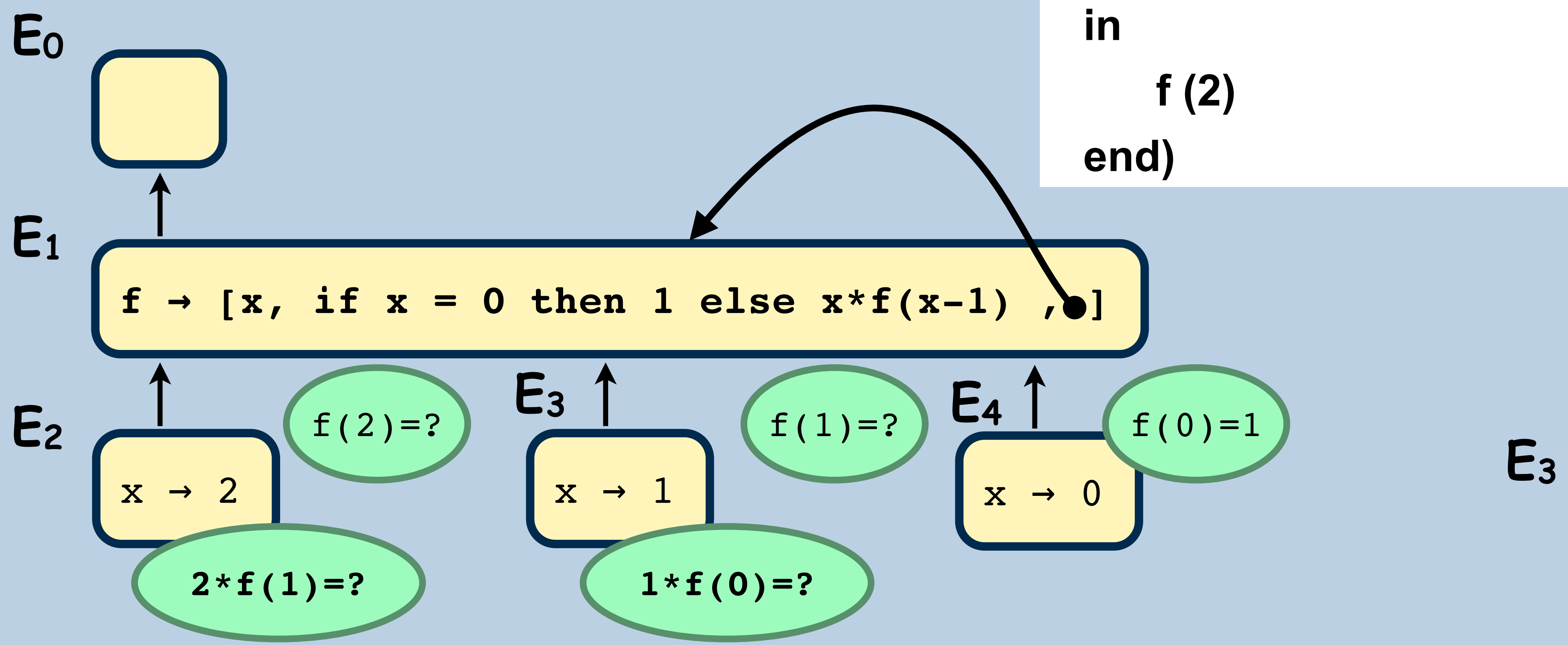
Example Evaluation

```
(def f =
  fun x -> if x = 0 then 1
              else x*f(x-1) end
  in
     f (2)
  end)
```



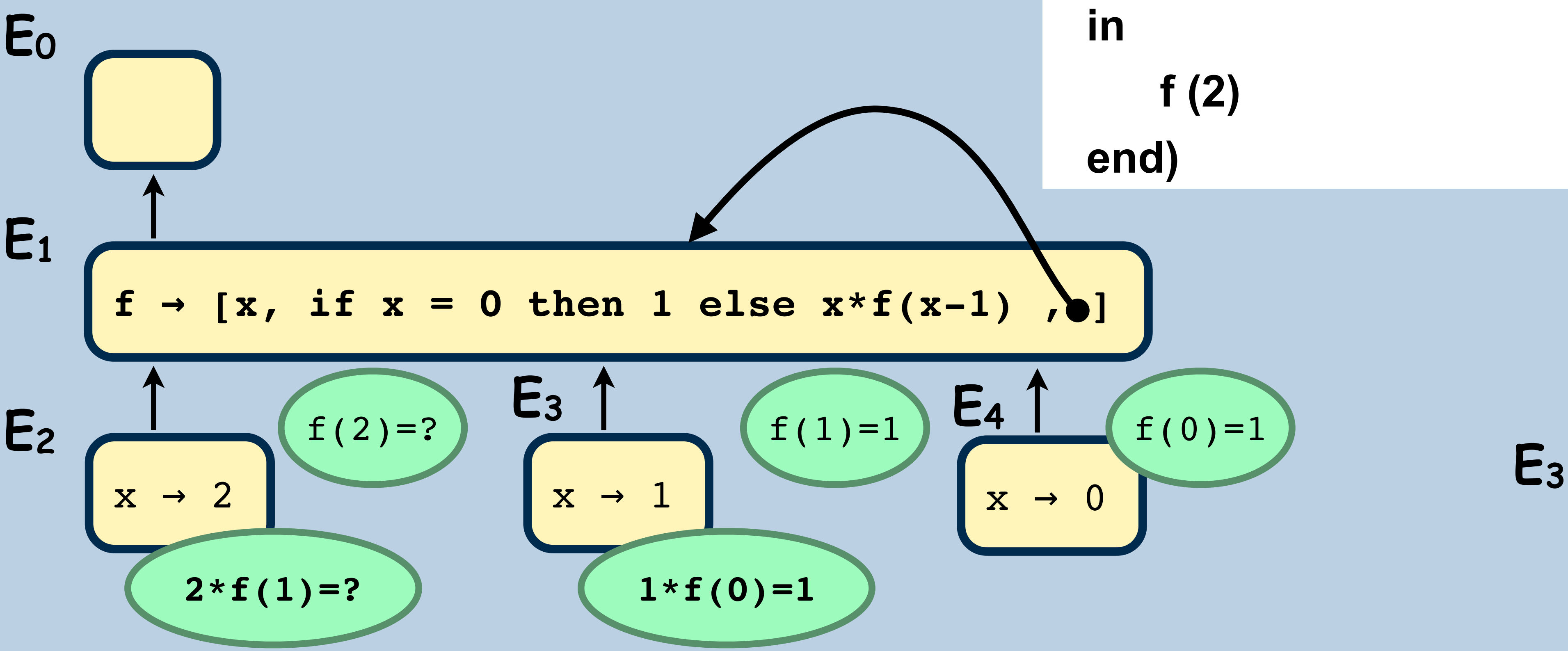
Example Evaluation

```
(def f =
  fun x -> if x = 0 then 1
                else x*f(x-1) end
  in
      f (2)
  end)
```



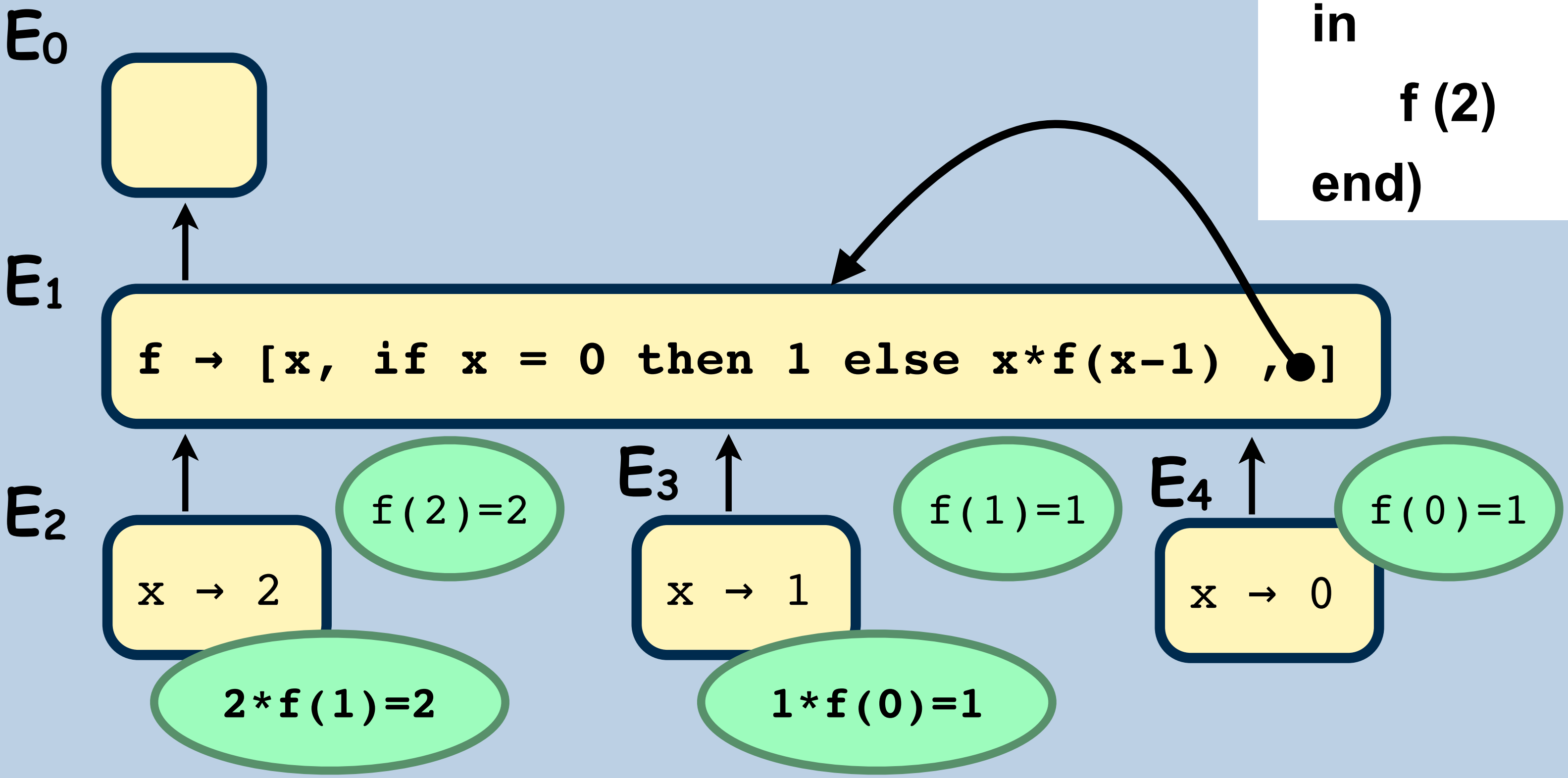
Example Evaluation

```
(def f =
  fun x -> if x = 0 then 1
              else x*f(x-1) end
  in
     f (2)
  end)
```



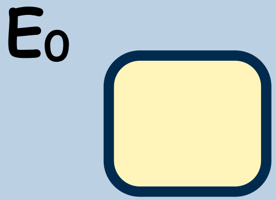
Example Evaluation

```
(def f =
  fun x -> if x = 0 then 1
              else x*f(x-1) end
  in
     f (2)
  end)
```



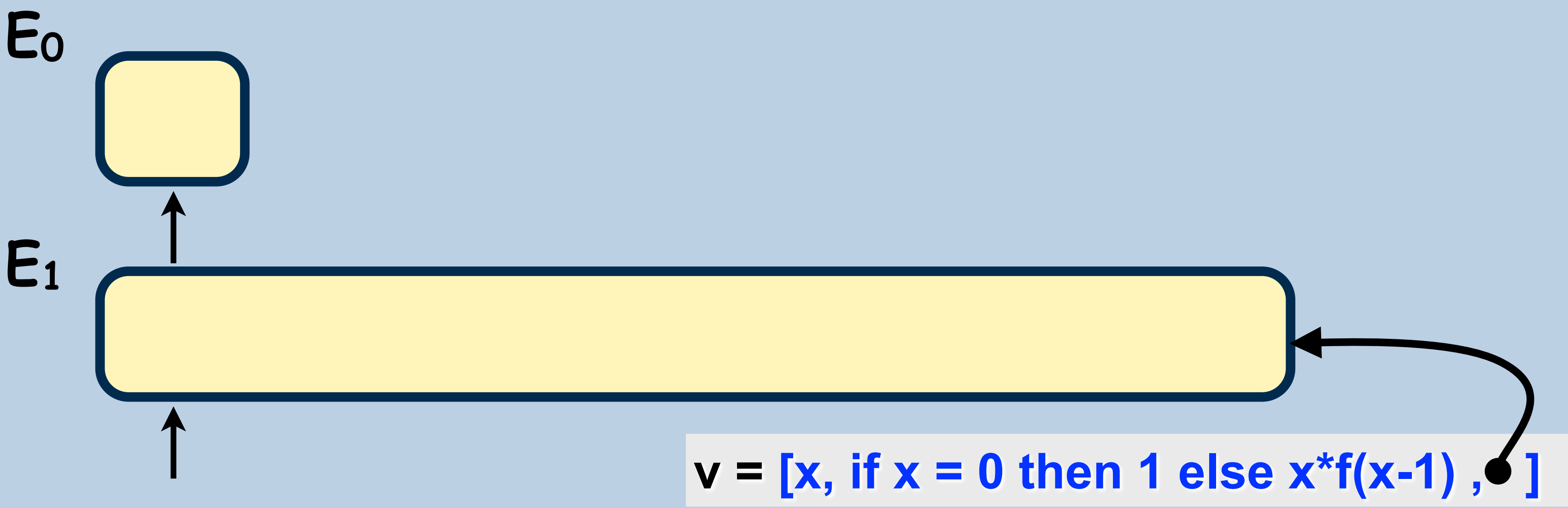
Recursion and Recursive Definitions

- Notice that the correct evaluation of the definition construct is already correct for recursive definitkons in our environment based interpreter;



Recursion and Recursive Definitions

- Notice that the correct evaluation of the definition construct is already correct for recursive definitkons in our environment based interpreter;

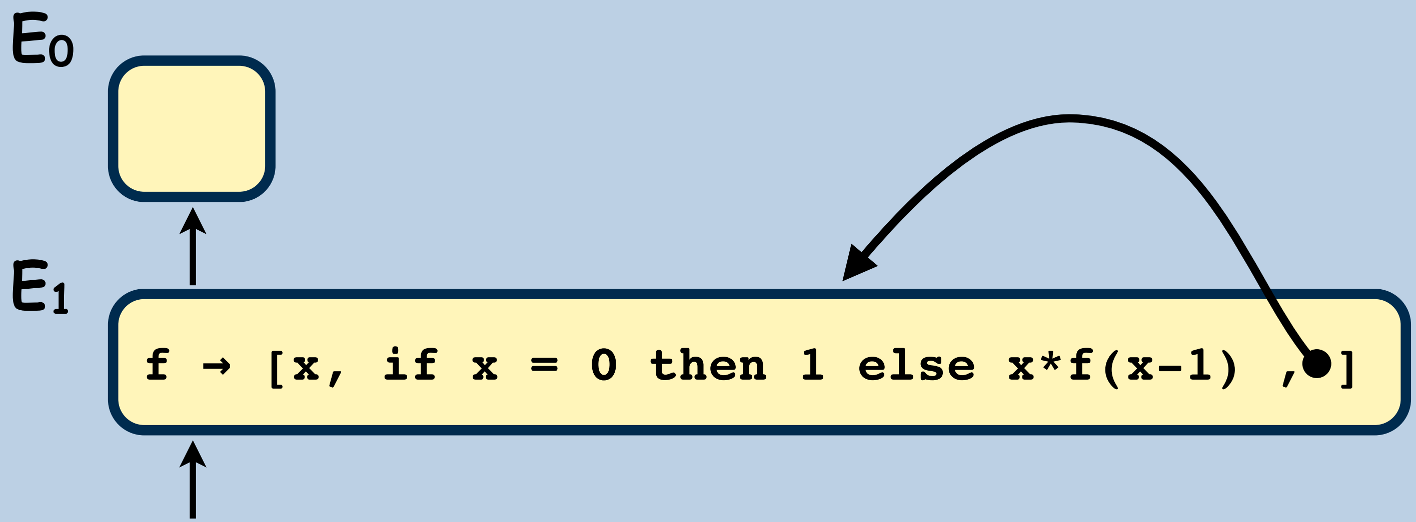


en = e.beginScope(); // creates new (empty) level

v = init.eval(en) // evaluate init in current level

Recursion and Recursive Definitions

- Notice that the correct evaluation of the definition construct is already correct for recursive definitkons in our environment based interpreter;




```
en = e.beginScope(); // creates new (empty) level
v = init.eval(en) // evaluate init in current level
en.assoc(“f”, v)
```