# Interpretação e Compilação (de Linguagens de Programação)

# Unidade 7: Abstracção funcional

Nas linguagens de programação existem várias formas de tornar código mais abstracto e reutilizável em diferentes contextos. A abstracção tem por objectivo aumentar o nível de detalhe com que se olha para um problema e pode ser conseguida ao nível da funcionalidade, dos dados, da iteração de colecções, através da hierarquização da informação, etc.

Todas as linguagens de programação modernas possuem uma ou mais destas formas de abstracção, e normalmente todas incluem suporte primitivo para a abstracção funcional por parametrização.

Abstracção por parameterização
Equivalência alfa
A linguagem CALCF
Semântica com substituição da linguagem CALCF
Algoritmo interpretador da linguagem CALCF com ambiente
Estratégias de resolução de nomes
Algoritmo compilador da linguagem CALCF
Declarações recursivas
Passagem de parâmetros (por valor, por nome)

# Abstracção por Parametrização

- As linguagens de programação têm um número finito de construções que podem ser usadas para representar um número infinito de computações.

- Podemos capturar grupos de computações semelhantes através da abstracção de alguns das suas componentes (as que variam).

- A abstracção é uma forma de enriquecer as linguagens de programação com novas operações, com um nível mais alto (manipulam entidades mais complexas como um todo) definidas a partir de outras construções da linguagem.

# Abstracção por Parametrização

- As linguagens de programação têm um número finito de construções que podem ser usadas para representar um número infinito de computações.
- Podemos capturar grupos de computações semelhantes através da abstracção de alguns das suas componentes (as que variam).
- A abstracção é uma forma de enriquecer as linguagens de programação com novas operações, com um nível mais alto (manipulam entidades mais complexas como um todo) definidas a partir de outras construções da linguagem.

```
1*1 + 2*2

1*1 + 1*2

2*2 + 1*1

1*2 + 2*1

3*3 + 4*4

2*2 + 3*2

5*5 + 7*7

2*2 + 9*9

1*1 + 2*2

3*1 + 1*2
```

# Abstracção por Parametrização

- As linguagens de programação têm um número finito de construções que podem ser usadas para representar um número infinito de computações.

- Podemos capturar grupos de computações semelhantes através da abstracção de alguns das suas componentes (as que variam).

- A abstracção é uma forma de enriquecer as linguagens de programação com novas operações, com um nível mais alto (manipulam entidades mais complexas como um todo) definidas a partir de outras construções da linguagem.

```
1*1 + 2*2

2*2 + 1*1

3*3 + 4*4

5*5 + 7*7

2*2 + 9*9

1*1 + 2*2
```

# Abstracção por Parametrização

- As linguagens de programação têm um número finito de construções que podem ser usadas para representar um número infinito de computações.

- Podemos capturar grupos de computações semelhantes através da abstracção de alguns das suas componentes (as que variam).

```
x*x + y*y
```

- A abstracção é uma forma de enriquecer as linguagens de programação com novas operações, com um nível mais alto (manipulam entidades mais complexas como um todo) definidas a partir de outras construções da linguagem.

# Abstracção por Parametrização

- É a possibilidade de definir entidades genéricas, introduzidas uma única vez e utilizáveis várias vezes num programa com diferentes instanciações de um conjunto de parâmetros definidos.
- É uma característica fundamental de todas as linguagens de programação modernas:
  - Funções / Procedimentos
  - Métodos
  - Tipos paramétricos
  - Classes paramétricas
  - etc...
- A parametrização e a abstracção são essenciais à modularidade dos programas, e em alguns casos, à expressividade computacional das linguagens.

# Parametrização

- É o mecanismo geral para declarar os nomes que se querem encarar como genéricos, destacando-os sintacticamente através de notação conveniente:

C: `int f(int x) { return x+1;}`

ML: `(fun x -> x+1)`

Lisp: `(lambda (x) (add x 1))`

Cálculo Lambda (Church): `λx.x`

- Tecnicamente, chama-se abstracção a uma construção sintáctica, explicitamente parametrizada num certo conjunto de nomes.

# Parametrização

- É o mecanismo geral para declarar os nomes que se querem encarar como genéricos, destacando-os sintacticamente através de notação conveniente:

C: `int f(int x) { return x+1;}`

ML: `(fun x -> x+1)`

Lisp: `(lambda (x) (add x 1))`

Cálculo Lambda (Church): `λx.x`

- Tecnicamente, chama-se abstracção a uma construção sintáctica, explicitamente parametrizada num certo conjunto de nomes.

# Parametrização

- É o mecanismo geral para declarar os nomes que se querem encarar como genéricos, destacando-os sintacticamente através de notação conveniente:

C: `int f(int x) { return x+1;}`

ML: `(fun x -> x+1)`

Lisp: `(lambda (x) (add x 1))`

Cálculo Lambda (Church): `λx.x`

- Tecnicamente, chama-se abstracção a uma construção sintáctica, explicitamente parametrizada num certo conjunto de nomes.

# Parametrização

- É o mecanismo geral para declarar os nomes que se querem encarar como genéricos, destacando-os sintacticamente através de notação conveniente:

C: `int f(int x) { return x+1;}`

ML: `(fun x -> x+1)`

Lisp: `(lambda (x) (add x 1))`

Cálculo Lambda (Church): `λx.x`

- Tecnicamente, chama-se abstracção a uma construção sintáctica, explicitamente parametrizada num certo conjunto de nomes.

# Parametrização

- É o mecanismo geral para declarar os nomes que se querem encarar como genéricos, destacando-os sintacticamente através de notação conveniente:

C: `int f(int x) { return x+1;}`

ML: `(fun x -> x+1)`

Lisp: `(lambda (x) (add x 1))`

Cálculo Lambda (Church): `λx.x`

- Tecnicamente, chama-se abstracção a uma construção sintáctica, explicitamente parametrizada num certo conjunto de nomes.

# Abstracção

- Uma abstracção é uma construção sintáctica constituída por dois elementos fundamentais:
  - As declarações dos seus parâmetros
  - O corpo da abstracção (uma qualquer construção da linguagem)

```
(x → x+y)
```

Parâmetros declarados: `x`; Corpo: `x+y`

```
(y → (x → x+y))
```

Parâmetros declarados: `y`; Corpo: `(x → x+y)`

```
(y,x → x+y)
```

Parâmetros declarados: `x,y`; Corpo: `x+y`

- Uma abstracção representa uma função anónima dos seus parâmetros...

# Observações

- As declarações dos parâmetros de uma abstracção são ocorrências ligantes dos nomes respectivos.
- O âmbito das declarações dos parâmetros é (exactamente) o corpo da abstracção.
- Os nomes livres duma abstracção são todos os nomes livres no seu corpo que não são parâmetros

$$\text{NomesLivres}((x_1,\ldots,x_n \rightarrow E) = \text{NomesLivres}(E) \setminus \{x_1,\ldots,x_n\}$$

exemplo:

$$\text{NomesLivres}((x \rightarrow x+y)) = \text{NomesLivres}(x+y) \setminus \{x\} = \{y\}$$

# Observações

- As declarações dos parâmetros de uma abstracção são ocorrências ligantes dos nomes respectivos.
- O âmbito das declarações dos parâmetros é (exactamente) o corpo da abstracção.
- Os nomes livres duma abstracção são todos os nomes livres no seu corpo que não são parâmetros

```
NomesLivres(x) = {x}
NomesLivres(E1 op E2) = NomesLivres(E1) U NomesLivres(E2)
NomesLivres(decl x = E1 in E2) = NomesLivres(E1) U NomesLivres(E2)\{x}
```

exemplos:

```
NomesLivres(x+decl y = x in x+y end) = {x}
NomesLivres(decl x = 1 in decl y = z in x+y end end) = {z}
```

# Equivalência-alfa ( $=_\alpha$ )

- Duas abstracções dizem-se alfa-equivalentes se uma pode ser obtida a partir da outra através da renomeação dos seus parâmetros, evitando conflitos com os seus nomes livres:

  $(x \rightarrow x+y) =_\alpha (z \rightarrow z+y)$

  $(x \rightarrow x+y) \neq_\alpha (y \rightarrow y+y)$

- Abstracções alfa-equivalentes são semanticamente equivalentes (são interpretadas como denotando a mesma função):

  $(x \rightarrow x+1) =_\alpha (z \rightarrow z+1)$

- Por isso, um interpretador pode traduzir os nomes dos parâmetros em algo mais conveniente e conhecido apenas localmente...

# O que denota uma abstracção?

- Uma abstracção é uma expressão sintáctica que denota uma certa função.
- Uma função é uma entidade semântica primitiva que suporta uma operação de aplicação, tal como um valor inteiro é uma entidade semântica primitiva que suporta a operação de adição, etc.
- Uma linguagem pode suportar funções mas não abstracções (exemplo: C, C++, Pascal)
- Várias linguagens suportam abstracções (ML, Smalltalk, Python, Javascript, Java (usando objectos anónimos))

# Abstracções vs. Declarações

- Em C e Pascal, as expressões que representam funções aparecem sempre no contexto de uma declaração:
  - `int f(int x) { return x+1; } (`*resto do programa*`)`
- Em ML, o mecanismo de declaração está separado do mecanismo de abstracção:
  - `let x=2 in (resto do programa)`
  - `let f = (fun x->x+1) in (resto do programa)`
  - `(map (fun x-> x+1) [1;2;3]) = [2;3;4]`
- Em ML, as funções são "cidadãs de primeira classe" (first-class citizens) ou seja, são tratadas como qualquer outro valor da linguagem. O mesmo não se passa com as funções em Pascal ou C.

# Abstracções (exemplos)

- Smalltalk
  - [ :x | E ] representa um bloco de programa, e é uma abstracção
  - 1 to: 10 do: [ :i | s ← s+i ] é a forma de fazer um ciclo...
- Abstracções em linguagens de objectos
  - Uma abstracção pode ser representada por um objecto que implementa um método "apply" (Function object)
  - Exemplo em Java...

```
Interface Function { int apply(int x); }
...
Function f = new Function{
       int apply(int x) {
           return x+1;
       }
}  // f  = λx. x+1
...
int v = f.apply(2) // v = (f 2)
```

# Abstracções (exemplos)

- As abstracções podem realizar-se não apenas sobre identificadores de valores, mas também sobre outras entidades, como por exemplo tipos.

- C++

```
template <class T> class Stack { int push(const &T); … }
```

- Java (5.0)

```
interface List<E> { void add(E x); Iterator<E> iterator(); }

interface Iterator<E> { E next(); boolean hasNext(); }

class LinkedList<E> implements List<E> { … }
```

# Exemplo: A Linguagem CALCF

- A linguagem CALCF estende a linguagem CALCI com funções, representadas por abstracções:

```
fun Id -> Expression end
```

- As funções podem ser aplicadas ao seu argumento, usando a expressão de aplicação:

```
Expression1 ( Expression2 )
```

- Semântica pretendida:
  Se ***Expression1*** denota uma função f, e ***Expression2*** denota um valor qualquer v, então ***Expression1*****(*****Expression2*****)** denota o valor que resulta de aplicar f a v.

# Exemplo: A Linguagem CALCF

- Um programa simples:

```
fun x -> x+2 end (4)
```

- Um outro exemplo:

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

- O mesmo exemplo numa sintaxe concreta tipo C:

```
function f(x){ return x+1;}
function g(y){ f(y)+2 }
{ ... x = g(2); return x+x; }
```

# A Linguagem CALCF

- Tipo de dados CALCF com os constructores:

```
num:   Integer → CALCF
add:   CALCF × CALCF → CALCF
mul:   CALCF × CALCF → CALCF
div:   CALCF × CALCF → CALCF
sub:   CALCF × CALCF → CALCF
id:    String → CALCF
decl:  String × CALCF × CALCF → CALCF
fun:   String × CALCF → CALCF
call:  CALCF × CALCF → CALCF
```

# Semântica de CALCF (1)

A função semântica I de CALCF:

$$I : CALCF \rightarrow RESULT$$

CALCF = conjunto dos programas fechados

RESULT = conjunto dos significados (denotações)

- Um significado pode ser um valor inteiro ou uma função (representada por uma abstracção):

$$RESULT = Integer \cup Abstraction \cup \{ error \}$$

# Resultados de CALCF

- Os significados de programas da linguagem CALCF podem ser apresentados como um tipo indutivo

- Tipo de dados RESULT com os constructores num e abstraction

```
num:         Integer → RESULT
abstraction: String × CALCF → RESULT
error:       void → RESULT
```

# Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

$$eval : CALCF \rightarrow RESULT$$

```
eval( num(n) )          ≜ n
eval( add(E1,E2) )      ≜ eval(E1) + eval(E2)
    …
eval( decl(s, E1, E2))  ≜ eval(subst(s, E2, E1))
```

# Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

$$eval : CALCF \rightarrow RESULT$$

```
eval( num(n) )          ≜ n
eval( add(E1,E2) )      ≜ eval(E1) + eval(E2)
    …
eval( decl(s, E1, E2))  ≜ eval(subst(s, E2, eval(E1)))
```

# Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

$$eval : CALCF \rightarrow RESULT$$

```
eval( num(n) )          ≜ n
eval( add(E1,E2) )      ≜ eval(E1) + eval(E2)
    …
eval( decl(s, E1, E2))  ≜ eval(subst(s, E2, eval(E1)))
```

calcular o valor primeiro
substitui depois...

# Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

$$\text{eval} : \text{CALCF} \rightarrow \text{RESULT}$$

```
eval( num(n) )          ≜ n
eval( add(E1,E2) )      ≜ eval(E1) + eval(E2)
    …
eval( decl(s, E1, E2))  ≜ eval(subst(s, E2, eval(E1)))
eval( fun(s, E) )       ≜ abstraction(s, E)
eval( call(E1, E2) )    ≜ [arg = eval(E2); fun = eval(E1);
                  if fun is abstraction(param, body)
                  then eval( subst(param, body, arg) )
                  else error ]
```

# Exemplos

# Exemplos

```
subst(x, x, 2*y ) = 2*y
```

# Exemplos

```
subst(x, x, 2*y ) = 2*y
```

```
subst(x, 2*4+(x+2), 2*y ) = 2*4+(2*y+2)
```

# Exemplos

```
subst(x, x, 2*y ) = 2*y
```

```
subst(x, 2*4+(x+2), 2*y ) = 2*4+(2*y+2)
```

```
subst(x, decl x = 1 in x+y end, 2*y ) = decl x = 1 in x+y end
```

# Exemplos

```
subst(x, x, 2*y ) = 2*y
```

```
subst(x, 2*4+(x+2), 2*y ) = 2*4+(2*y+2)
```

```
subst(x, decl x = 1 in x+y end, 2*y ) = decl x = 1 in x+y end
```

```
subst(x, decl z = 1 in x+z end, 2*y ) = ?
```

# Exemplos

```
subst(x, x, 2*y ) = 2*y
```

```
subst(x, 2*4+(x+2), 2*y ) = 2*4+(2*y+2)
```

```
subst(x, decl x = 1 in x+y end, 2*y ) = decl x = 1 in x+y end
```

```
subst(x, decl z = 1 in x+z end, 2*y ) = decl z = 1 in 2*y+z end
```

# Exemplos

```
subst(x, x, 2*y ) = 2*y
```

```
subst(x, 2*4+(x+2), 2*y ) = 2*4+(2*y+2)
```

```
subst(x, decl x = 1 in x+y end, 2*y ) = decl x = 1 in x+y end
```

```
subst(x, decl z = 1 in x+z end, 2*y ) = decl z = 1 in 2*y+z end
```

```
subst(x, decl y = 1 in x+y end, 2*y ) = ?
```

# Exemplos

```
subst(x, x, 2*y ) = 2*y
```

```
subst(x, 2*4+(x+2), 2*y ) = 2*4+(2*y+2)
```

```
subst(x, decl x = 1 in x+y end, 2*y ) = decl x = 1 in x+y end
```

```
subst(x, decl z = 1 in x+z end, 2*y ) = decl z = 1 in 2*y+z end
```

```
subst(x, decl y = 1 in x+y end, 2*y ) = decl y = 1 in 2*y+y end
```

# Exemplos

```
subst(x, x, 2*y ) = 2*y
```

```
subst(x, 2*4+(x+2), 2*y ) = 2*4+(2*y+2)
```

```
subst(x, decl x = 1 in x+y end, 2*y ) = decl x = 1 in x+y end
```

```
subst(x, decl z = 1 in x+z end, 2*y ) = decl z = 1 in 2*y+z end
```

```
subst(x, decl y = 1 in x+y end, 2*y ) = decl y = 1 in 2*y+y end
```

**A ligação da ocorrência do identificador y que foi inserido na expressão à sua declaração foi "capturada"!!**

# Definição da Função Subst

```
subst(s, num(n), F )      ≜ num(n);
subst(s, id(s), F )       ≜ F;
subst(s, add(E1,E2),F )   ≜ add( subst(s, E1, F), subst(s,E2,F));
…

subst(s, decl(s, E1, E2), F) ≜ [ /* caso s = s' */
                    G = subst(s, E1, F);
                    decl(s, G, E2);   ]

subst(s, decl(s', E1, E2), F) ≜ [ /* caso s ≠ s' */
                     G = subst(s, E1, F);
                     newid = fresh_identifier();
                     E2' = subst(s', E2, newid);
                     decl(newid, G, subst(s, E2', F)); ]
```

# Definição da Função Subst

```
subst(s, num(n), F )      ≜ num(n);
subst(s, id(s), F )       ≜ F;
subst(s, add(E1,E2),F )   ≜ add( subst(s, E1, F), subst(s,E2,F));
…

subst(s, decl(s, E1, E2), F) ≜ [ /* caso s = s' */
                    G = subst(s, E1, F);
                    decl(s, G, E2);   ]

subst(s, decl(s', E1, E2), F) ≜ [ /* caso s ≠ s' */
                     G = subst(s, E1, F);
                     newid = fresh_identifier();
                     E2' = subst(s', E2, newid);
                     decl(newid, G, subst(s, E2', F)); ]
```

A renomeação do identificador local evita a captura ilegal de ocorrências livres do identificador s' em F

# Definição da Função Subst

```
subst(s, call(E1, E2) ) ≙  call( subst(s, E1,F), subst(s, E2,F))

subst(s, fun(s', E), F) ≙ /* caso s = s' */
            fun (s, E)

subst(s, fun(s', E), F) ≙ [/* caso s ≠ s' */
            newid = fresh_identifier();
            E' = subst(s', E, newid);
            fun(newid, subst(s, E', F))]
```

# Definição da Função Subst

```
subst(s, call(E1, E2) ) ≙  call( subst(s, E1,F), subst(s, E2,F))

subst(s, fun(s', E), F) ≙ /* caso s = s' */
            fun (s, E)

subst(s, fun(s', E), F) ≙ [/* caso s ≠ s' */
            newid = fresh_identifier();
            E' = subst(s', E, newid);
            fun(newid, subst(s, E', F))]
```

A renomeação do identificador local evita a captura ilegal de ocorrências livres do identificador s' em F

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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


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

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

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

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

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

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

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

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

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

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

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

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

```
(fun x -> x+(fun y->y+1)(x))(2)
```

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

```
(fun x -> x+(fun y->y+1)(x))(2)
```

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

```
(fun x -> x+(fun y->y+1)(x))(2)
```

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

```
(2+(fun y->y+1)(2))
```

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

```
(2+(fun y->y+1)(2))
```

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

```
(2+(2+1))
```

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

```
(2+3)
```

# Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

(5)

# Quiz

- Quais os passos da avaliação do programa P segundo a semântica com substituição?

- ```
  decl y = 3 in
    decl x = 2*y in
      decl f = (fun y -> y+x) in
        decl g = (fun x -> (fun h -> x+h(x))
          in (g(2))(f)
  ```

```
eval( num(n) )           ≙ n
eval( add(E1,E2) )       ≙ eval(E1) + eval(E2)
   …
eval( decl(s, E1, E2))   ≙ eval(subst(s, E2, eval(E1)))
eval( fun(s, E) )        ≙ abstraction(s, E)
eval( call(E1, E2) )     ≙ [arg = eval(E2); fun = eval(E1);
              if fun is abstraction(param, body)
              then eval( subst(param, body, arg) )
              else error ]
```
```

# Semântica de CALCF (2)

A função semântica I de CALCF:

$$I : CALCF \times ENV \rightarrow RESULT$$

CALCF = conjunto dos programas abertos

ENV = conjunto dos ambientes válidos

RESULT = conjunto dos significados (denotações)

- Um significado pode ser um valor inteiro ou uma função(representada por uma abstracção):

$$RESULT = Integer \cup Abstraction \cup \{ error \}$$

# Interpretador de CALCF (2)

- Algoritmo eval(E, env) para calcular a denotação de uma expressão E de CALCF:

$$eval : CALCF \times ENV \rightarrow RESULT$$

```
eval( num(n) , env)     ≜ n
eval( id(s) , env)      ≜ env.Find(s)
…
eval( fun(s, B), env )     ≜ abstraction(s, B)
eval( call(E1, E2) , env ) ≜ fun = eval(E1, env )
                if fun is abstraction(s, B) then
                [ env'=env.BeginScope();
                  env'.Assoc(s, eval(E2, env )) ;
                  val = eval(B, env' );
                  env'.EndScope();
                  val ]
                else error
```

# Interpretador de CALCF (2)

- Avaliação do programa:

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

# Interpretador de CALCF (2)

- Avaliação do programa:

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

eval(P1,0) =

# Interpretador de CALCF (2)

- Avaliação do programa:

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

eval(P1,0) =

eval(P2, [f=abs(x,x+1)]) =

# Interpretador de CALCF (2)

- Avaliação do programa:

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

eval(P1,0) =

eval(P2, [f=abs(x,x+1)]) =

eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =

# Interpretador de CALCF (2)

- Avaliação do programa:

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

eval(P1,0) =

eval(P2, [f=abs(x,x+1)]) =

eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =

eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =

# Interpretador de CALCF (2)

- Avaliação do programa:

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

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(g, [g=abs(y,f(y)+2), f=abs(x,x+1) ]) = abs(y,f(y)+2)

# Interpretador de CALCF (2)

- Avaliação do programa:

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

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(f(y)+2, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1) ]) =

# Interpretador de CALCF (2)

- Avaliação do programa:

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

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(f(y)+2, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(f, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1) ]) = abs(x,x+1)

# Interpretador de CALCF (2)

- Avaliação do programa:

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

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(f(y)+2, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(x+1, [x=2, g=abs(y,f(y)+2), f=abs(x,x+1) ]) = 3

# Interpretador de CALCF (2)

- Avaliação do programa:

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

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(f(y)+2, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1) ]) = 5

# Interpretador de CALCF (2)

- Avaliação do programa:

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

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1) ]) =
eval(x+x, [x=5, g=abs(y,f(y)+2), f=abs(x,x+1) ]) = 10

# Interpretador de CALCF (2)

- Outro exemplo:

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

E = [ g=abs(x,x+f(x)); f=abs(y,y+x); x=1]

eval(g(2),E) =

eval(x+f(x), [x=2; g=abs(x,x+f(x)),f=abs(y, y+x)]) =

2+eval(f(x), [x=2; g=abs(x,x+f(x)),f=abs(y, y+x) ]) =

2+eval(y+x, [y=2;x=2;g=abs(x,x+f(x)),f=abs(y, y+x) ]) =

2+2+2 = 6

- Seguindo a sua intuição, qual é o valor deste programa?

# Interpretador de CALCF (2)

- Outro exemplo:

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

E = [ g=abs(x,x+f(x)); f=abs(y,y+x); x=1]

eval(g(2),E) =

eval(x+f(x), [x=2; g=abs(x,x+f(x)),f=abs(y, y+x)]) =

2+eval(f(x), [x=2; g=abs(x,x+f(x)),f=abs(y, y+x) ]) =

2+eval(y+x, [y=2;x=2;g=abs(x,x+f(x)),f=abs(y, y+x) ]) =

2+2+2 = 6

- O valor do programa deveria ser 5 ! O que falhou???

# Resolução dinâmica de nomes

- A semântica simplificada (2) que definimos para a linguagem CALCF adopta a "regra dinâmica" de resolução de nomes (dynamic scoping).
- Segundo esta regra, os valores dos nomes **não locais** são interpretados no contexto da chamada da função, em vez do contexto da definição.
- Historicamente, algumas linguagens de programação (ex: Lisp, JavaScript 1.0) adoptaram a resolução dinâmica de nomes, por ser de implementação mais simples.
- Actualmente, é considerado um mecanismo indesejável e até incorrecto (por não respeitar o princípio da substituição), apesar de às vezes "dar jeito" interpretar nomes não locais no contexto da chamada (por exemplo: "hostname").

# Princípio da substitutividade

- O valor de qualquer expressão permanece inalterado sempre que nela se substitui uma subexpressão por outra expressão com o mesmo significado / valor.
- A semântica da linguagem CALCF viola este princípio, pois os dois programa seguintes têm valores diferentes:

```
decl x=1 in
  decl f = (fun y→y+x) in
    decl g = (fun x→x+f(x))
      in g(2)
```

```
decl x=1 in
  decl f = (fun y→y+1) in
    decl g = (fun x→x+f(x))
      in g(2)
```

# Resolução estática de nomes

- Todas as linguagens de programação modernas adoptam a regra da resolução estática de identificadores (static scoping).

- Segundo esta regra, os valores dos identificadores livres que ocorram no corpo de abstracções são interpretados no contexto em que as abstracções ocorrem (na definição das funções), e não no contexto da chamada.

- Para implementar esta semântica de forma eficiente é necessário usar um domínio de resultados mais rico, em que as funções são representadas por entidades chamadas "fechos".

# Semântica de CALCF (3)

- A (nova) função semântica I de CALCF:

$$I : CALCF \times ENV \rightarrow RESULT$$

CALCF = conjunto dos programas abertos

ENV = conjunto dos ambientes

RESULT = conjunto dos significados (denotações)

Os resultados podem ser valores inteiros, fechos (uma abstracção + um ambiente), ou um erro.

$$RESULT = Integer \cup Closure \cup \{ error \}$$

# Resultados de CALCF

- Os significados de programas da linguagem CALCF podem ser apresentados como um tipo indutivo

Tipo de dados RESULT com os constructores num e closure

```
num:      Integer → RESULT
closure:  String × CALCF × ENV → RESULT
error:    void → RESULT
```

Um fecho representa uma função através de um triplo contendo o parâmetro, o corpo, e o ambiente que regista os valores dos nomes livres no corpo

Assim, ao contrário de uma abstracção, um fecho é efectivamente um valor "fechado" (não depende de nenhum nome externo).

# Ambiente "mutável"

- Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável:

```
Environ BeginScope()
```

  - Cria um novo nível **vazio**, onde serão colocadas as ligações de um novo âmbito local.
  - Não pode existir mais que uma ligação para um mesmo identificador no mesmo nível.

```
Environ EndScope()
```

  - Devolve o ambiente no estado anterior à última operação BeginScope().
  - Mas **não destrói** o último nível pois podem existir no contexto de execução fechos que o referem!

# Interpretador de CALCF (3)

- Algoritmo eval(E, env) para calcular o valor de uma expressão E de CALCF:

$$eval : CALCF \times ENV \rightarrow RESULT$$

```
eval( fun(s, B), env )      ≙ closure(s, env, B)
eval( call(E1, E2) , env ) ≙ fun = eval(E1, env )
                if fun is closure(s, envc, B) then
                [ envloc = envc.BeginScope();
                  envloc.Assoc(s, eval(E2, env )) ;
                  val = eval(B, envloc );
                  envc = envloc.EndScope();
                  val ]
                else error
```

# Avaliação de Funções

- Exemplo: avaliar o seguinte programa, na semântica usando ambientes e fechos.

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

# Exemplo

$E_0$

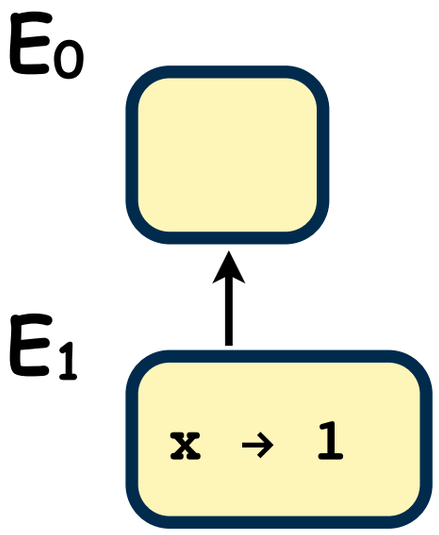
```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo

$E_0$

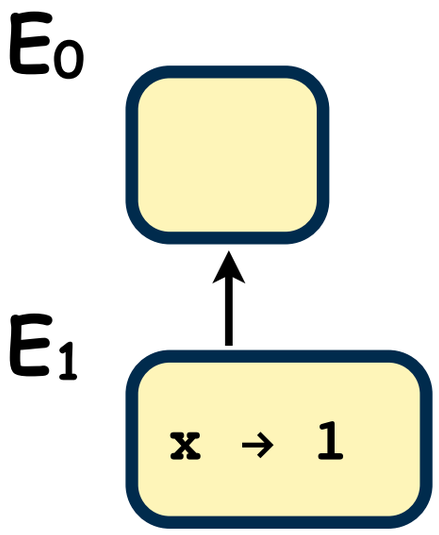
```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo



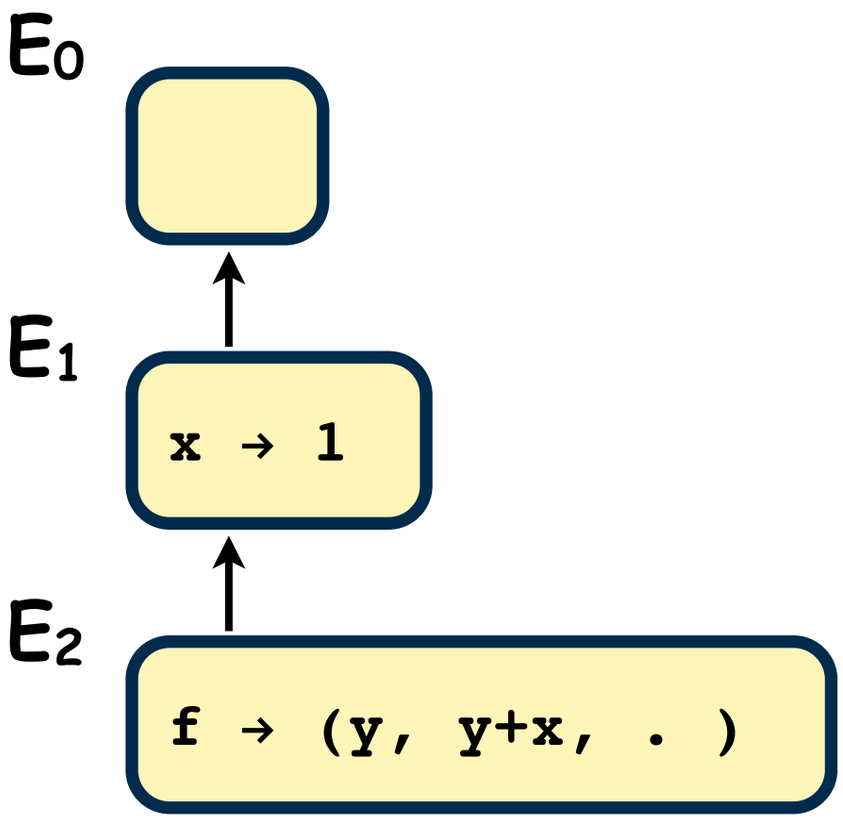

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo



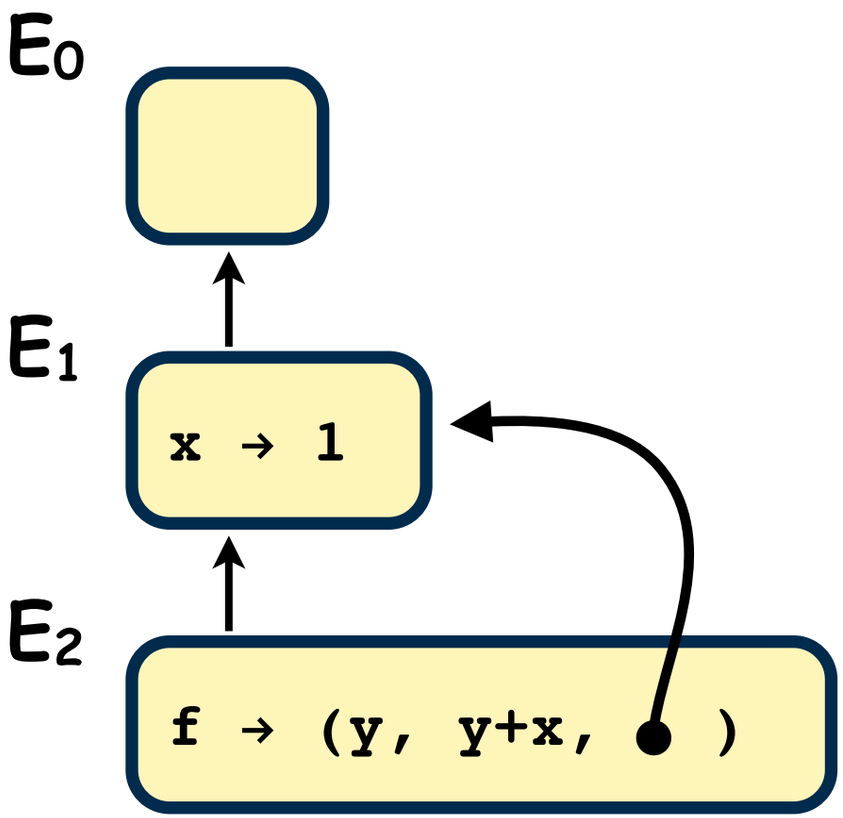

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo



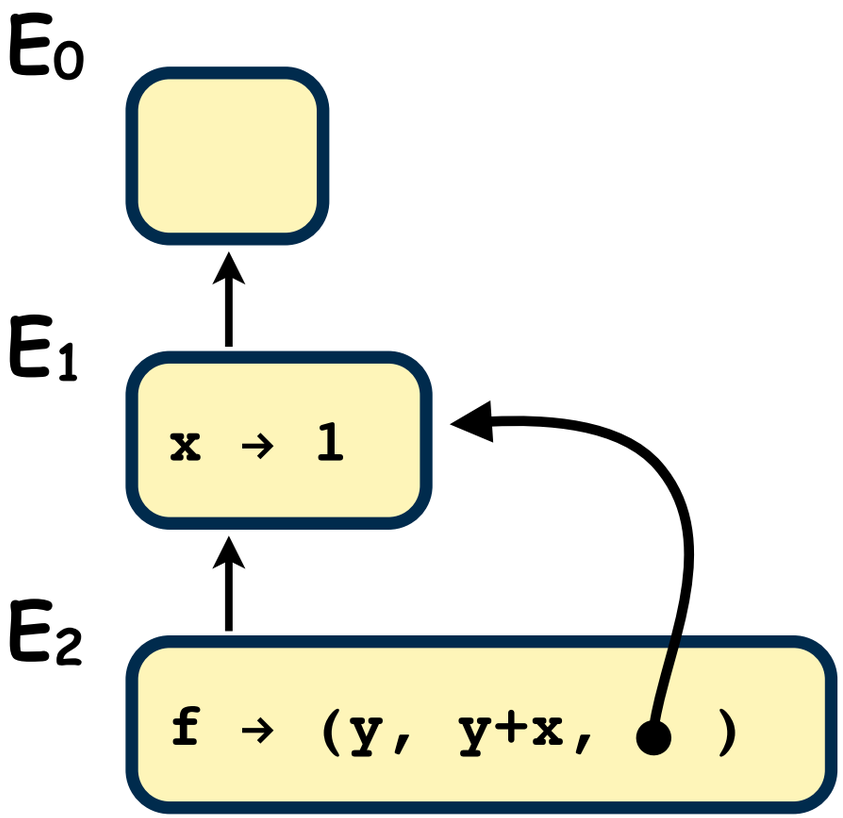

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo



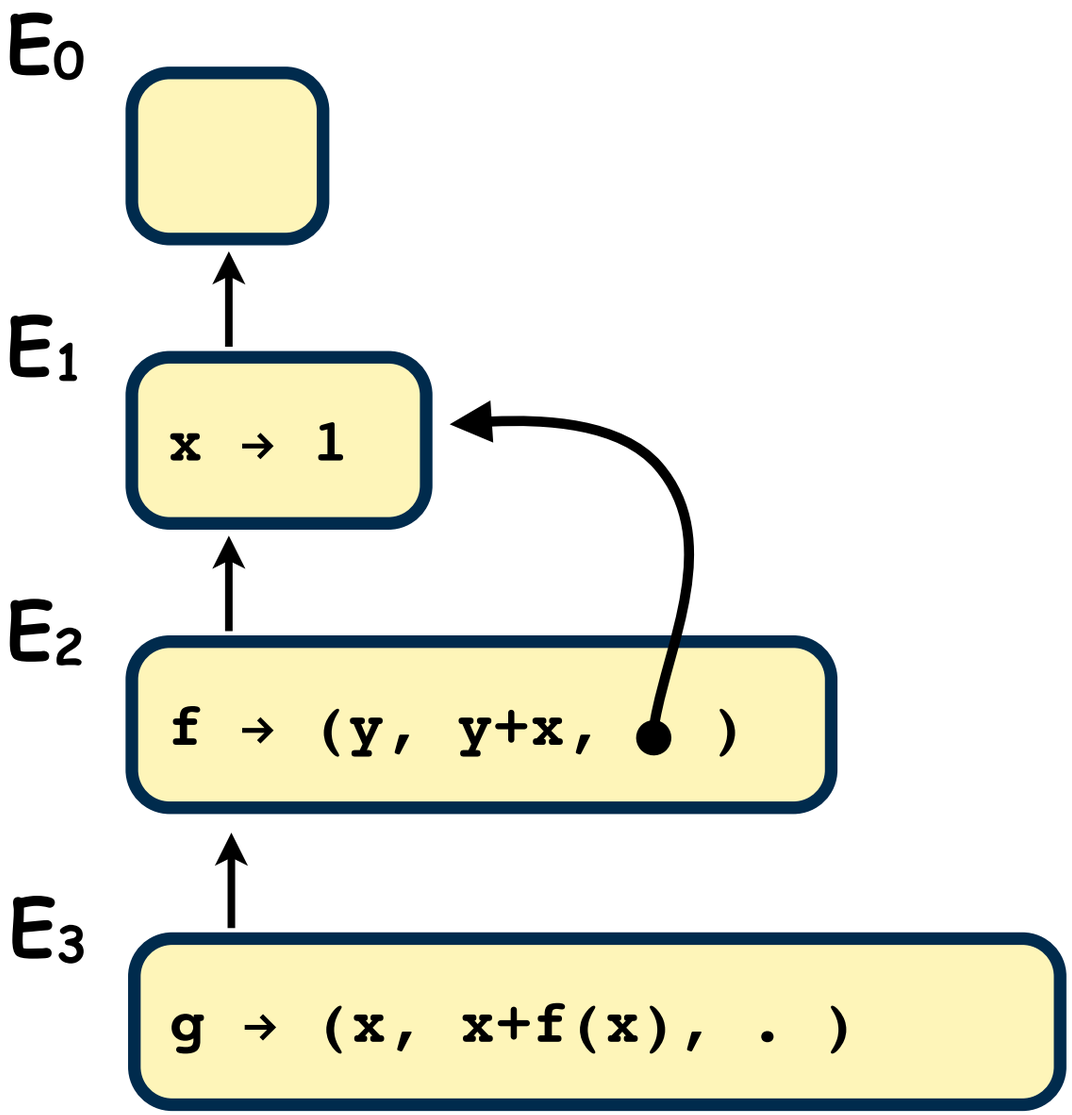

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo



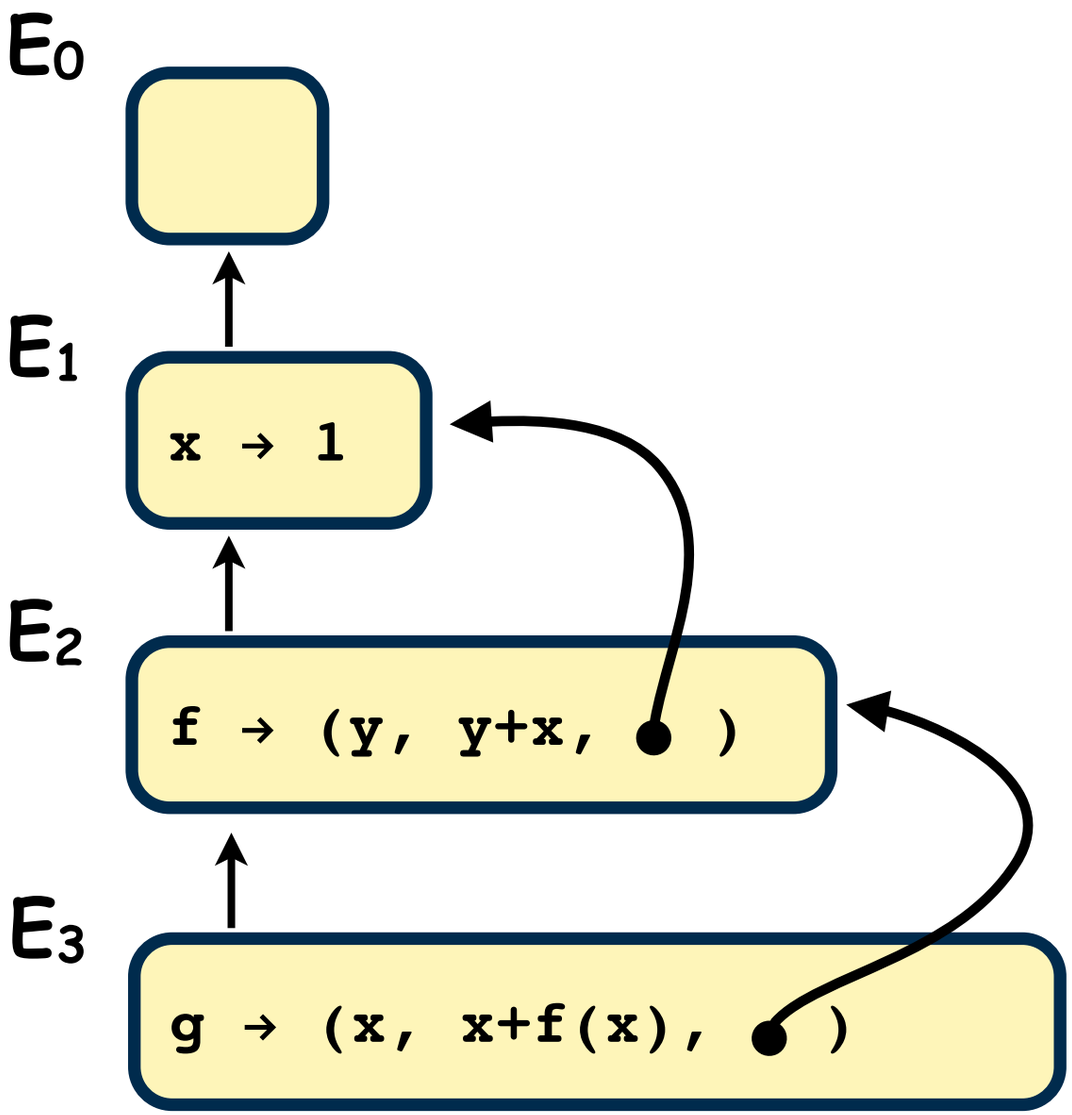

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo



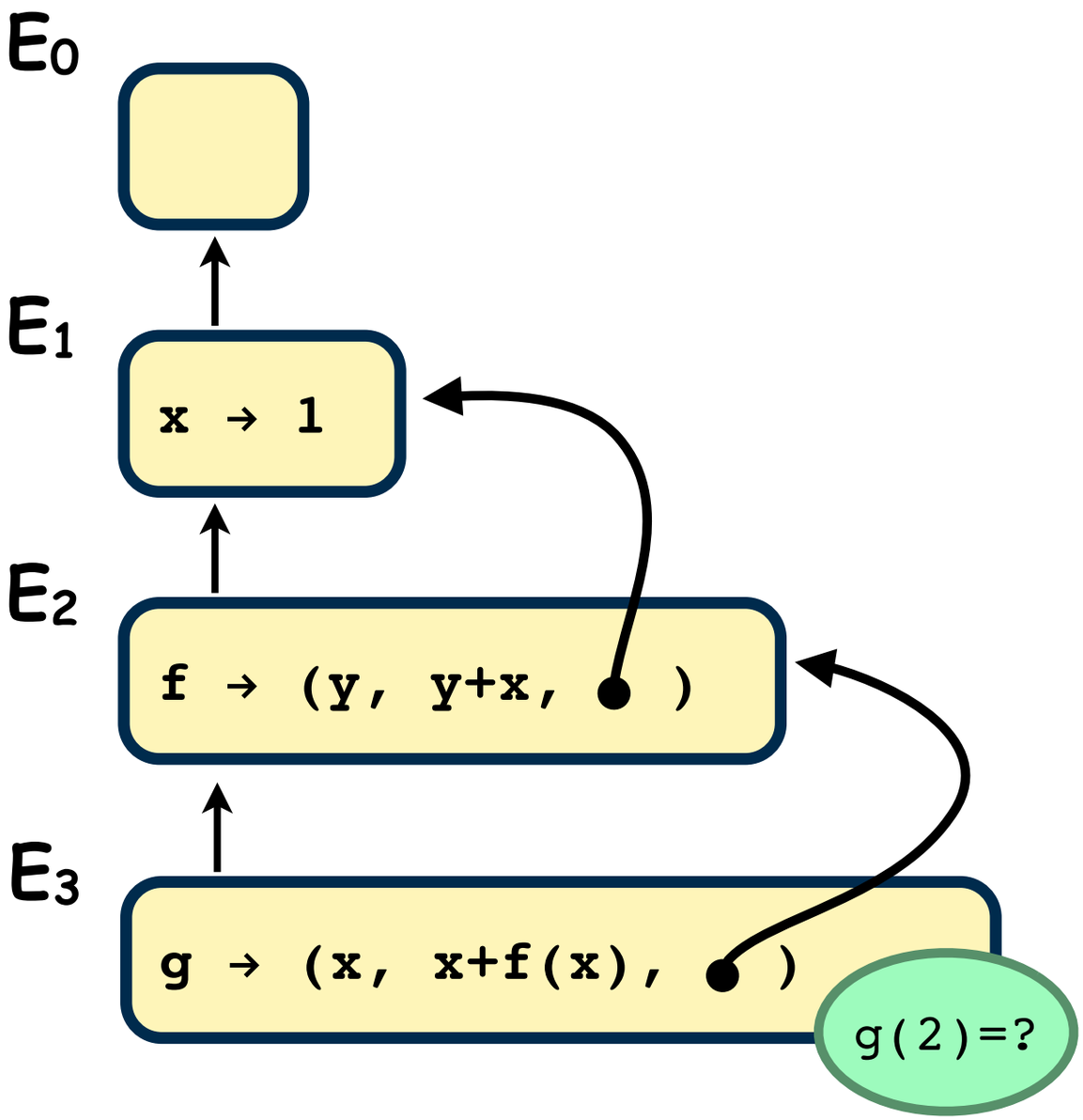

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo



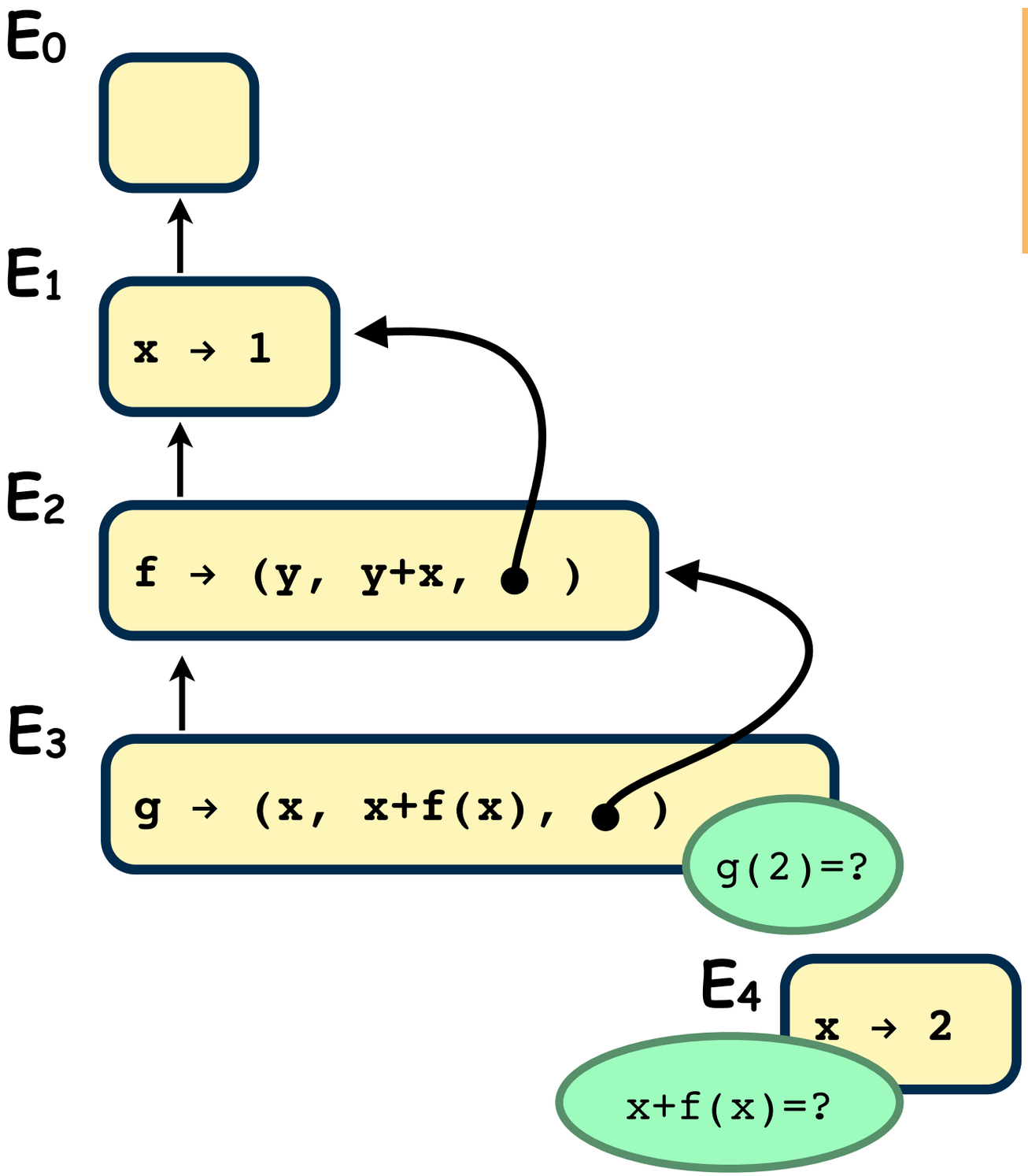

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo



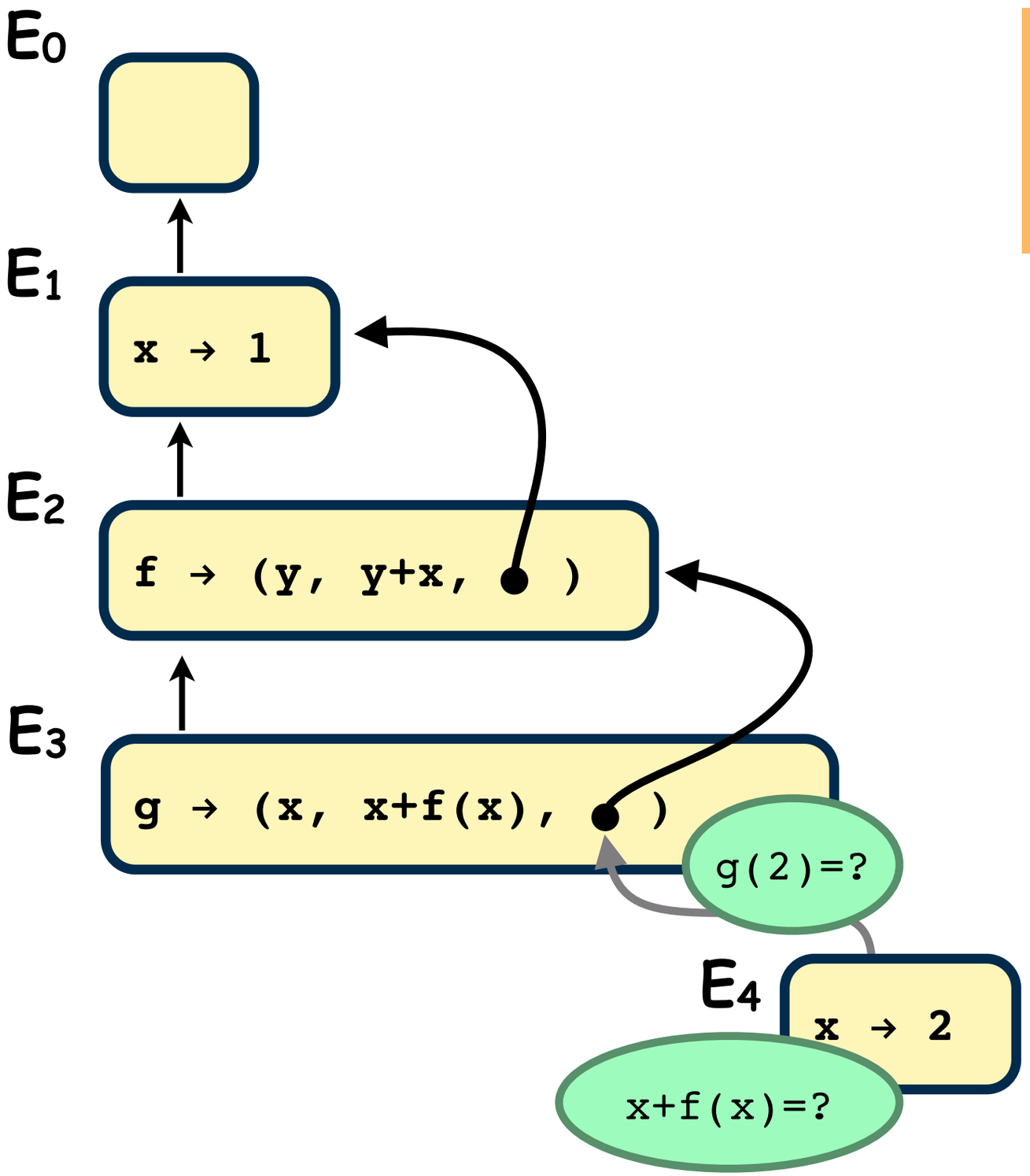

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo



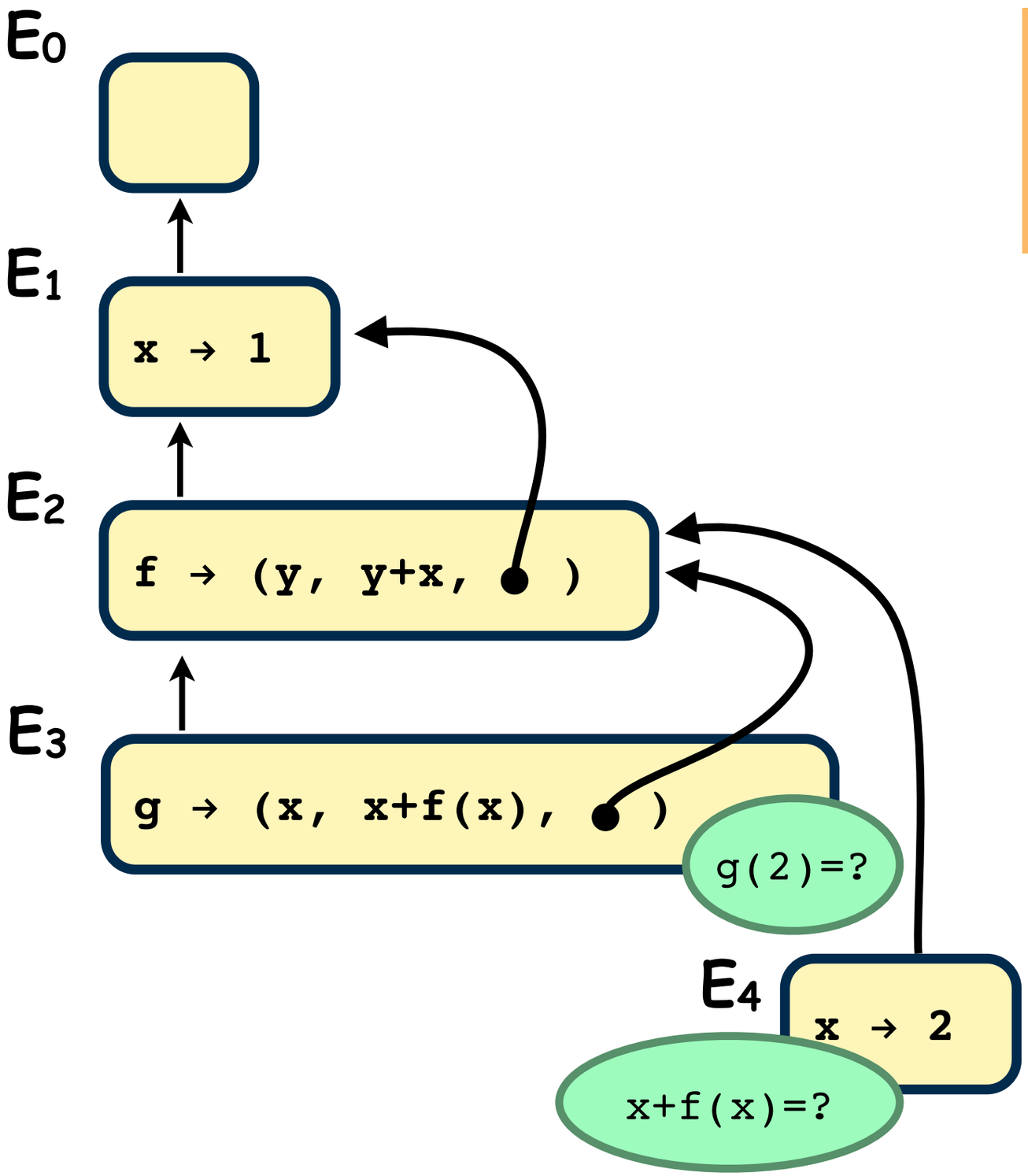

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo



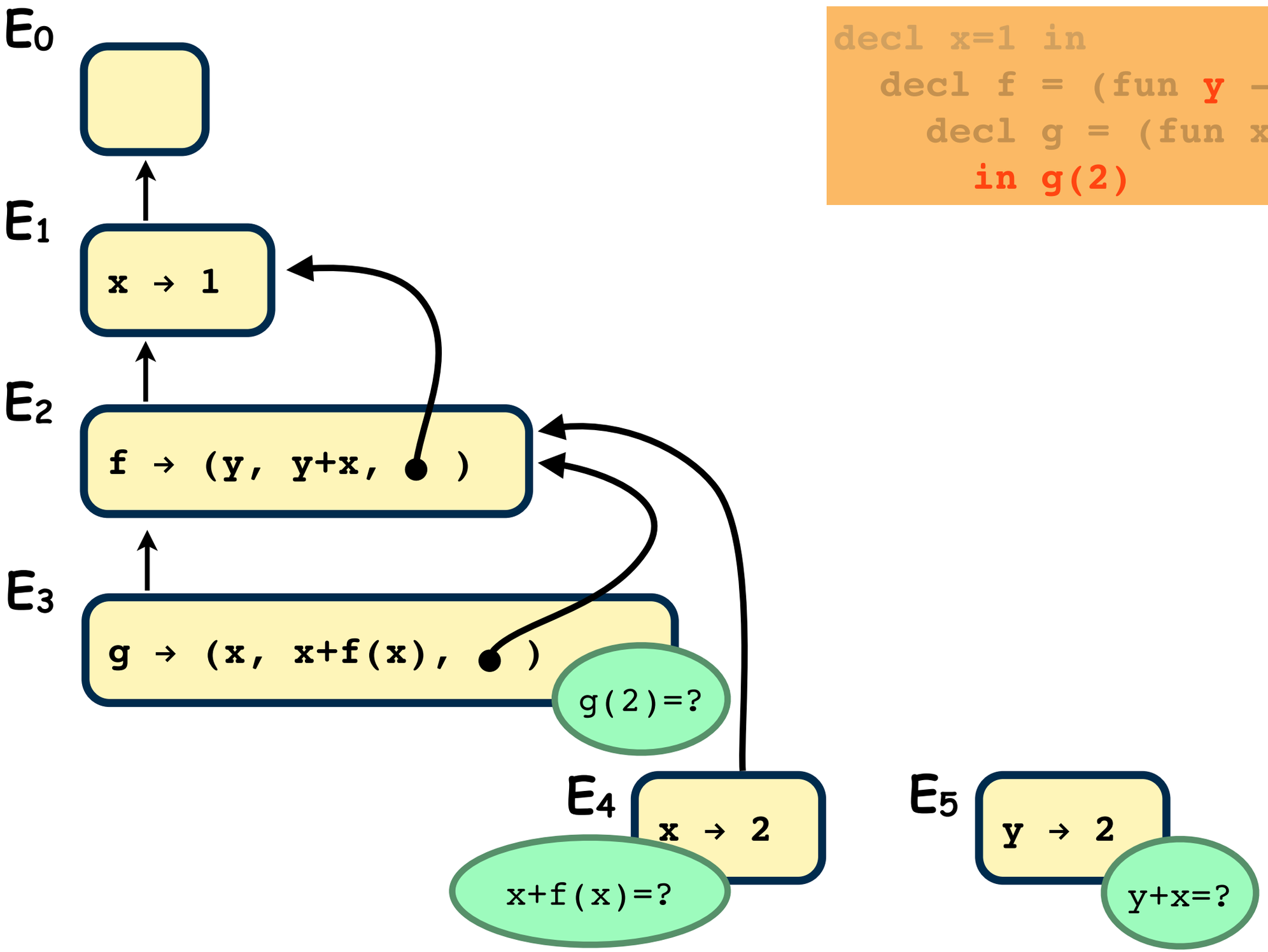

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo



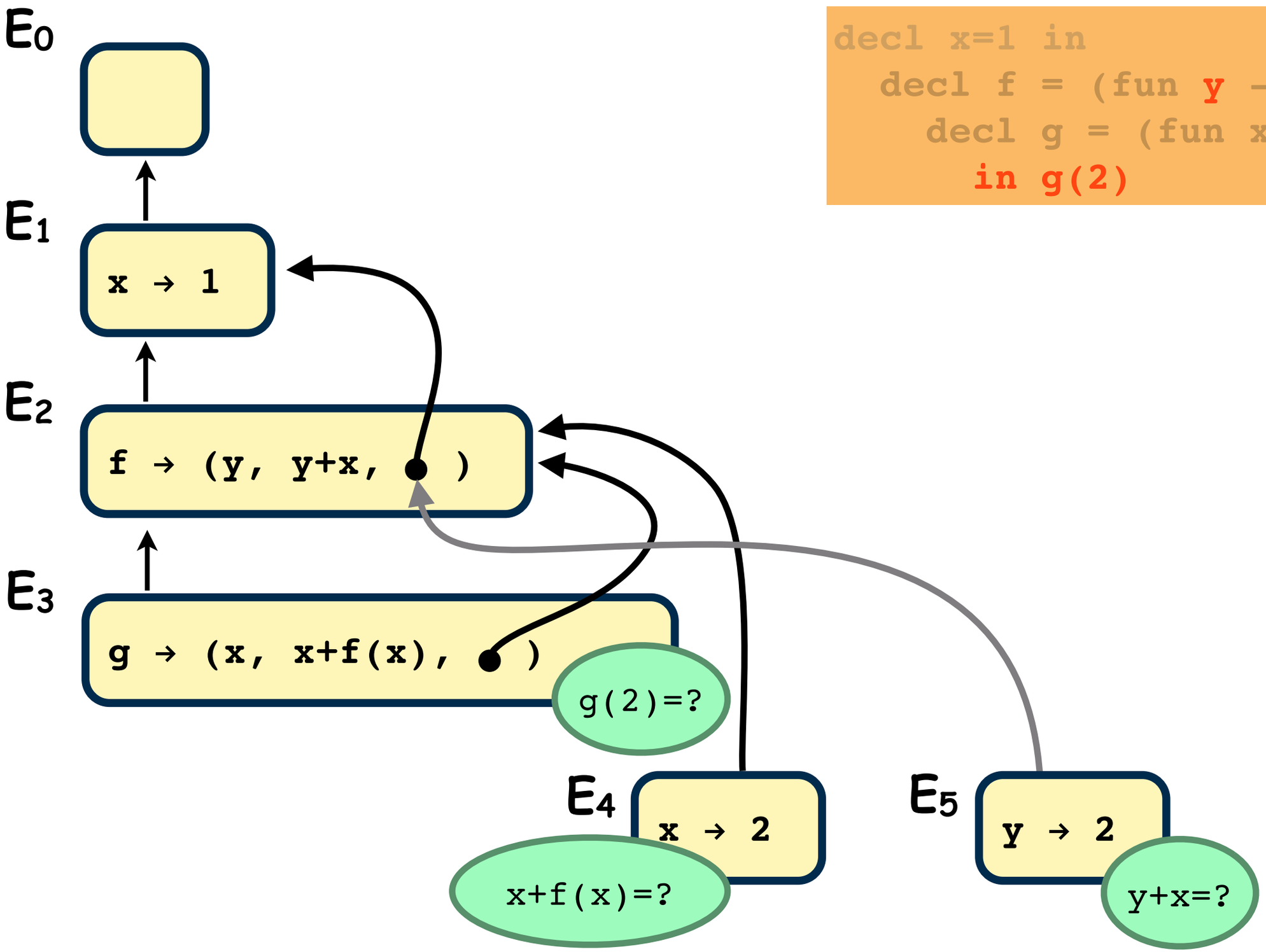

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

# Exemplo




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

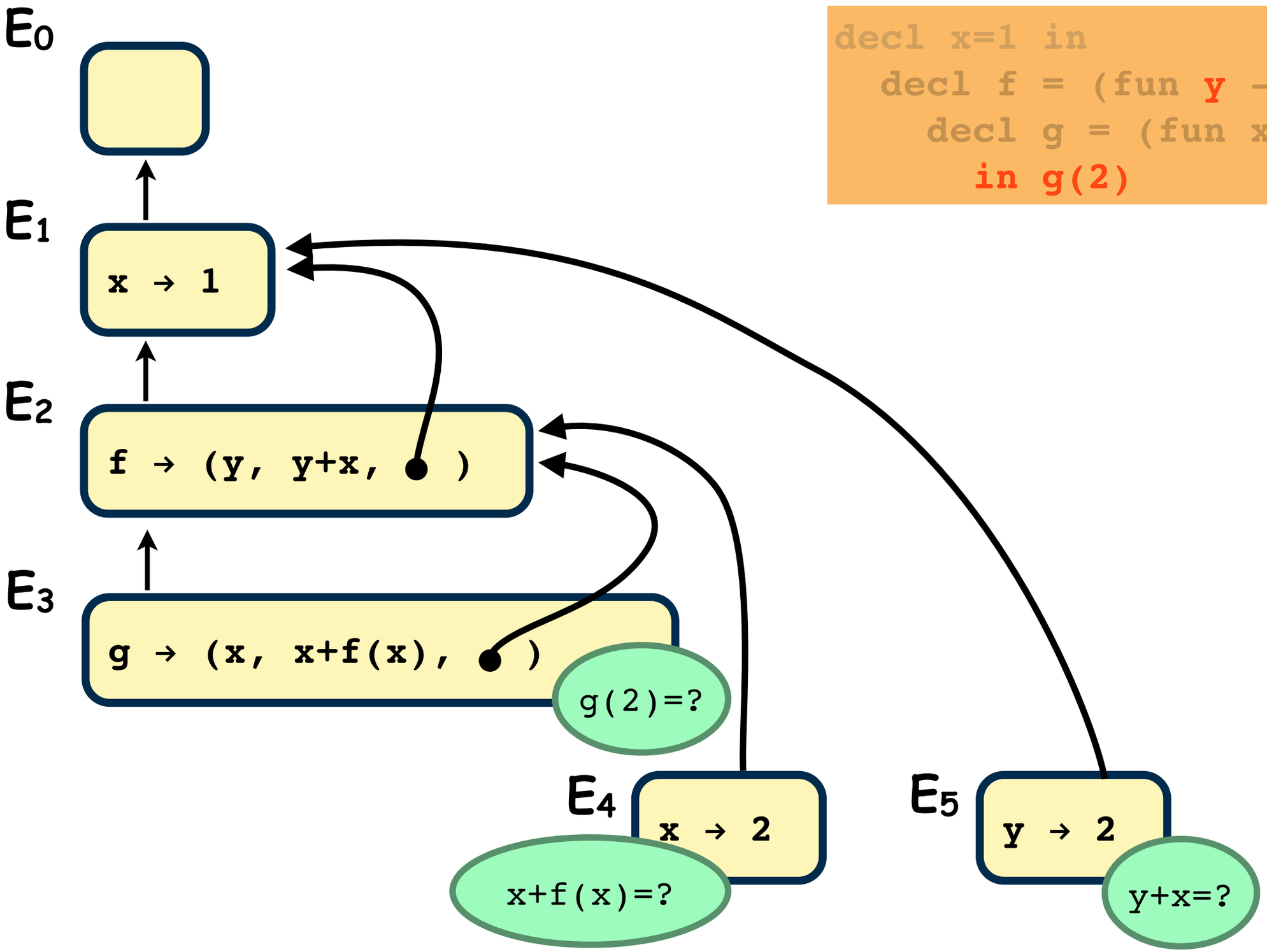
# Exemplo




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

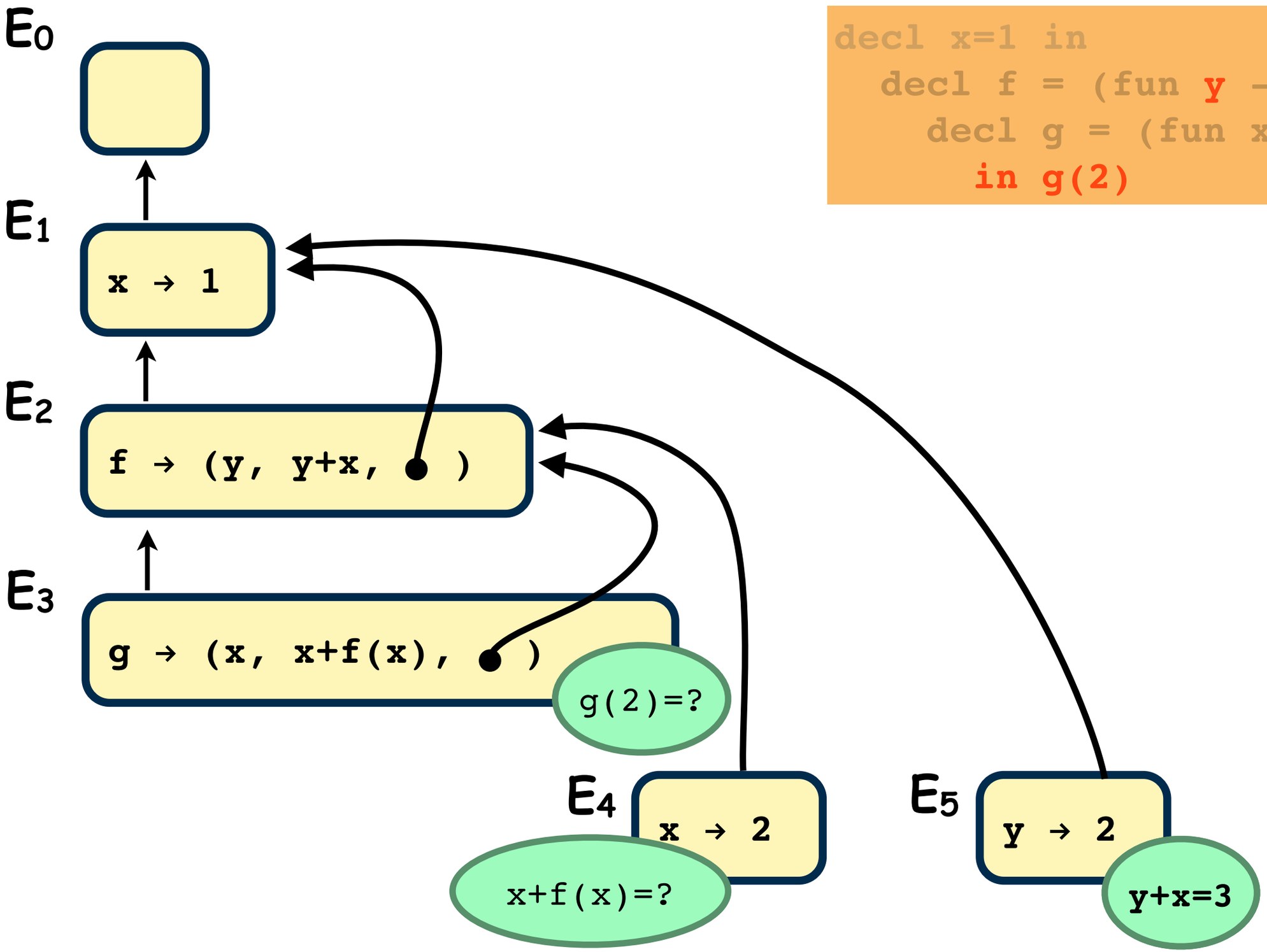
# Exemplo

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

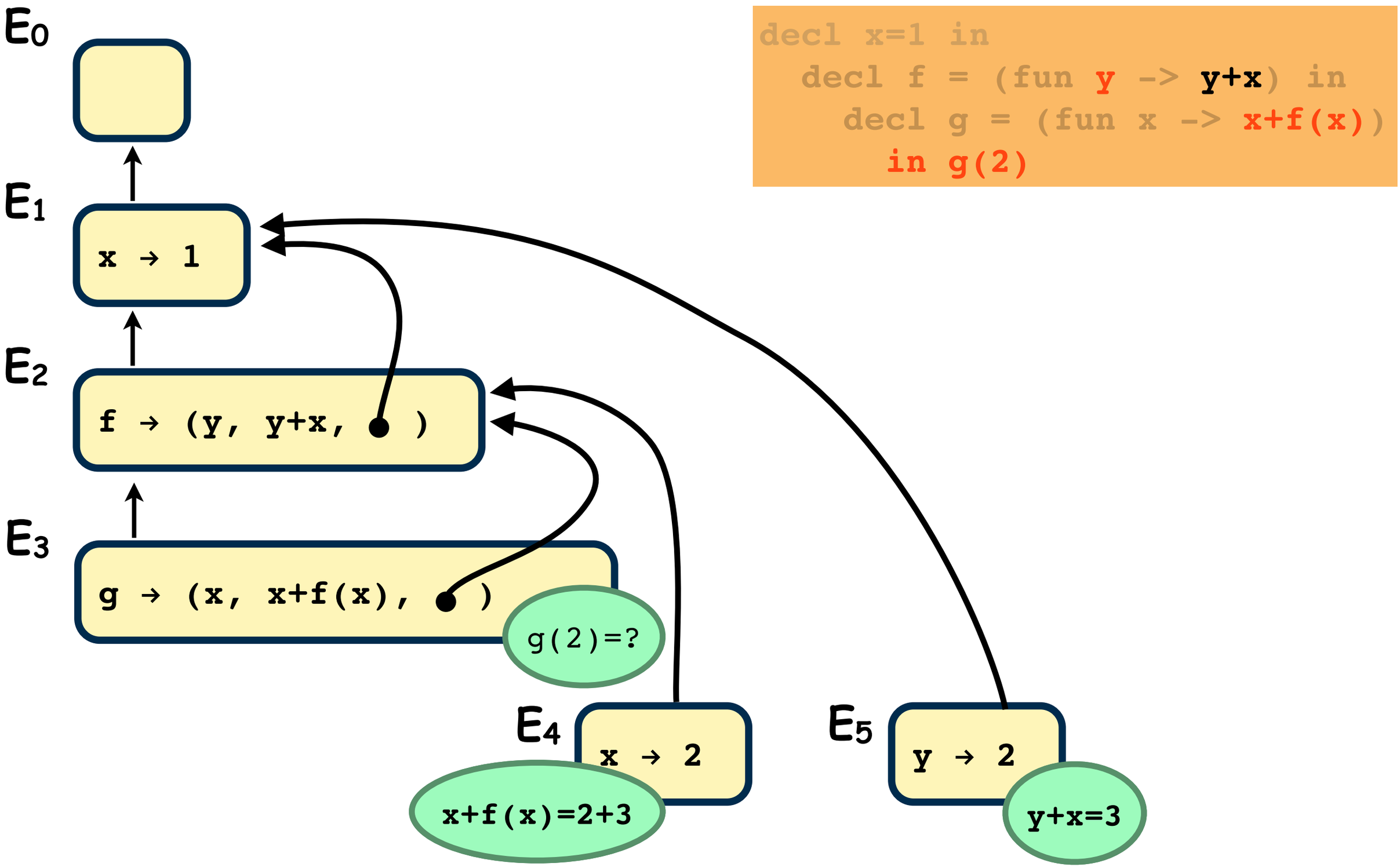


# Exemplo

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



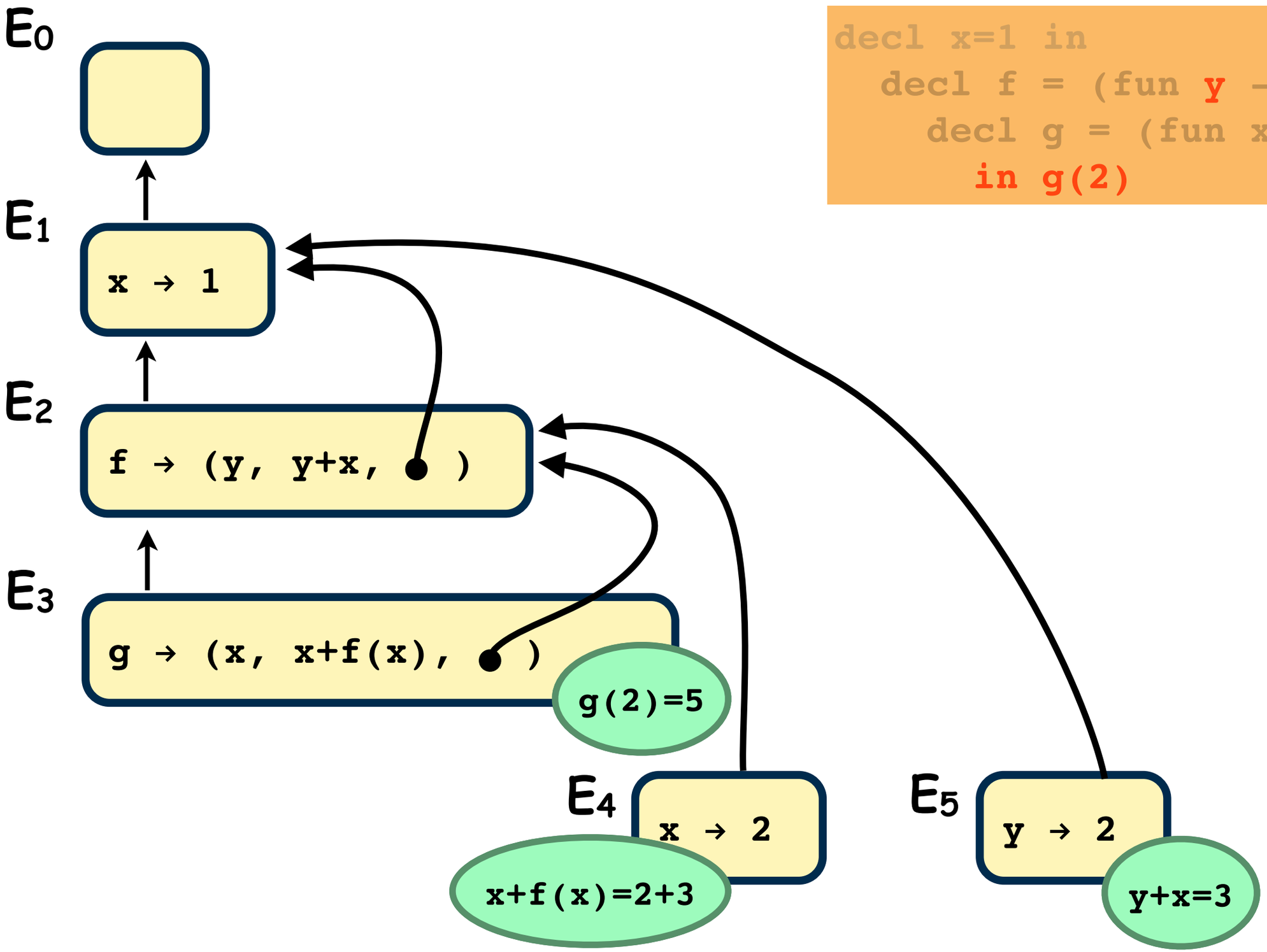
# Exemplo




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

# Exemplo

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



# Avaliação de Funções

- Exemplo: avaliar o seguinte programa, na semântica usando ambientes e fechos.

```
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)
```

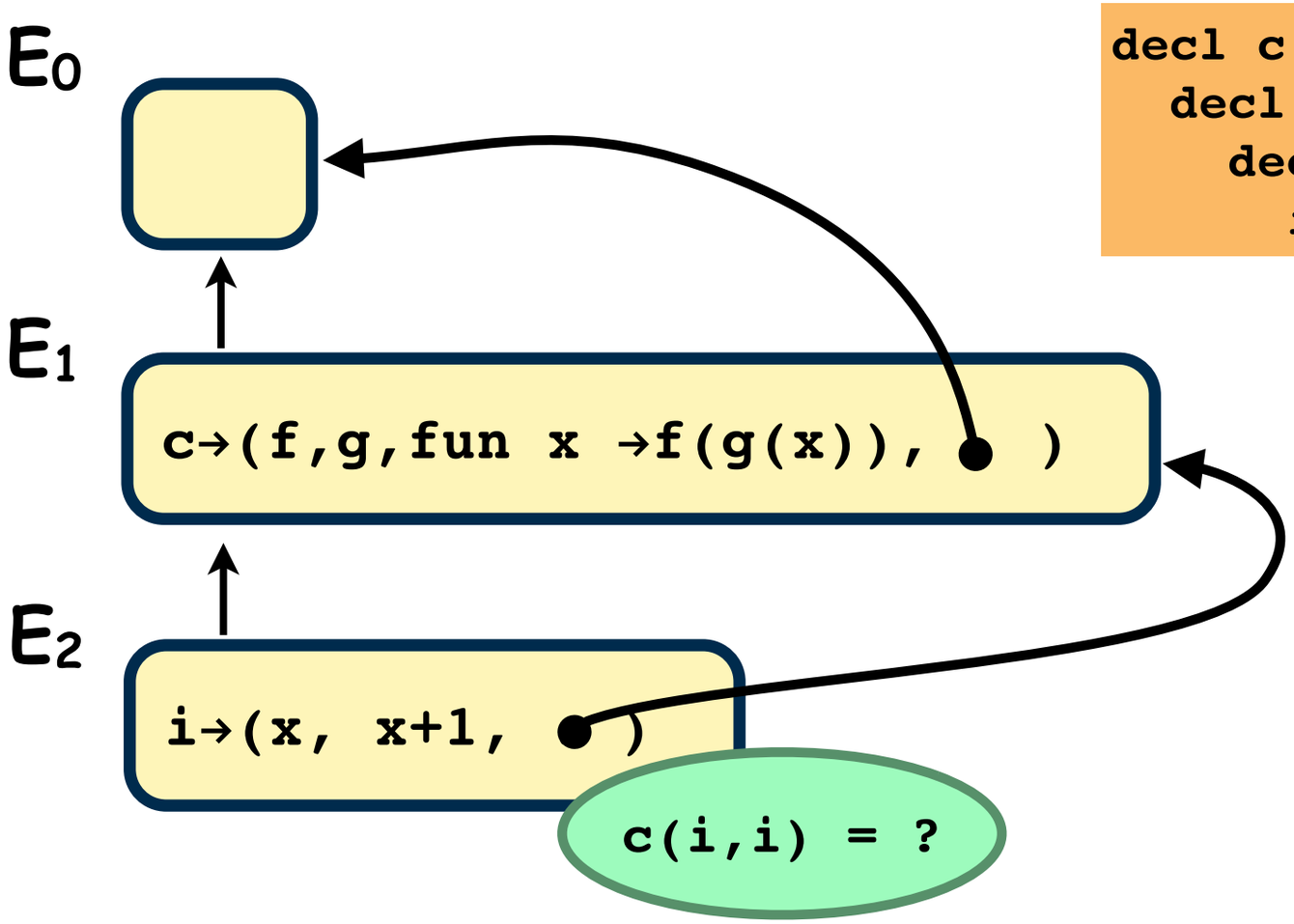
# Avaliação de Funções

- Exemplo: avaliar o seguinte programa, na semântica usando ambientes e fechos.

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

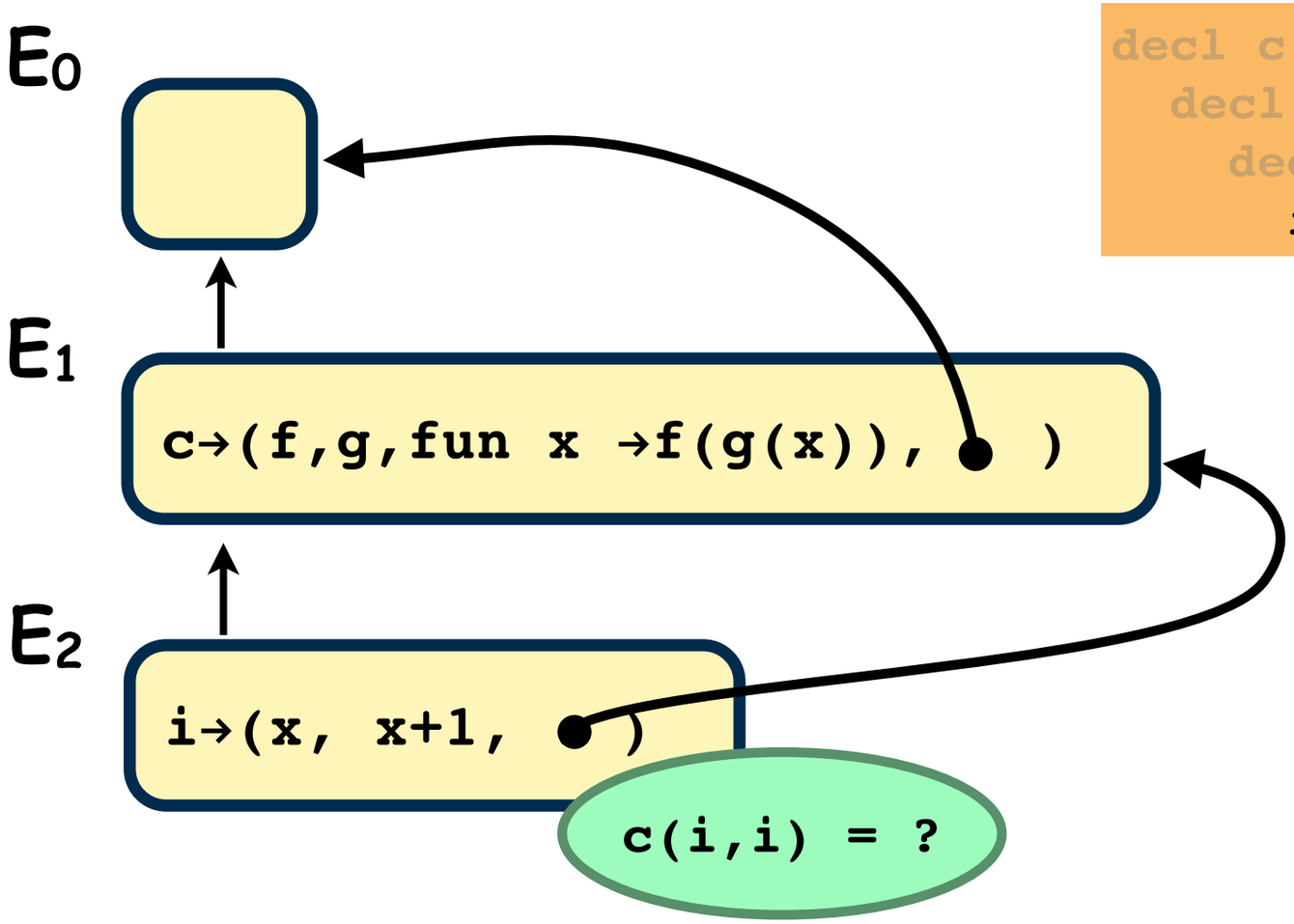
# Avaliação de Funções

```
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)
```

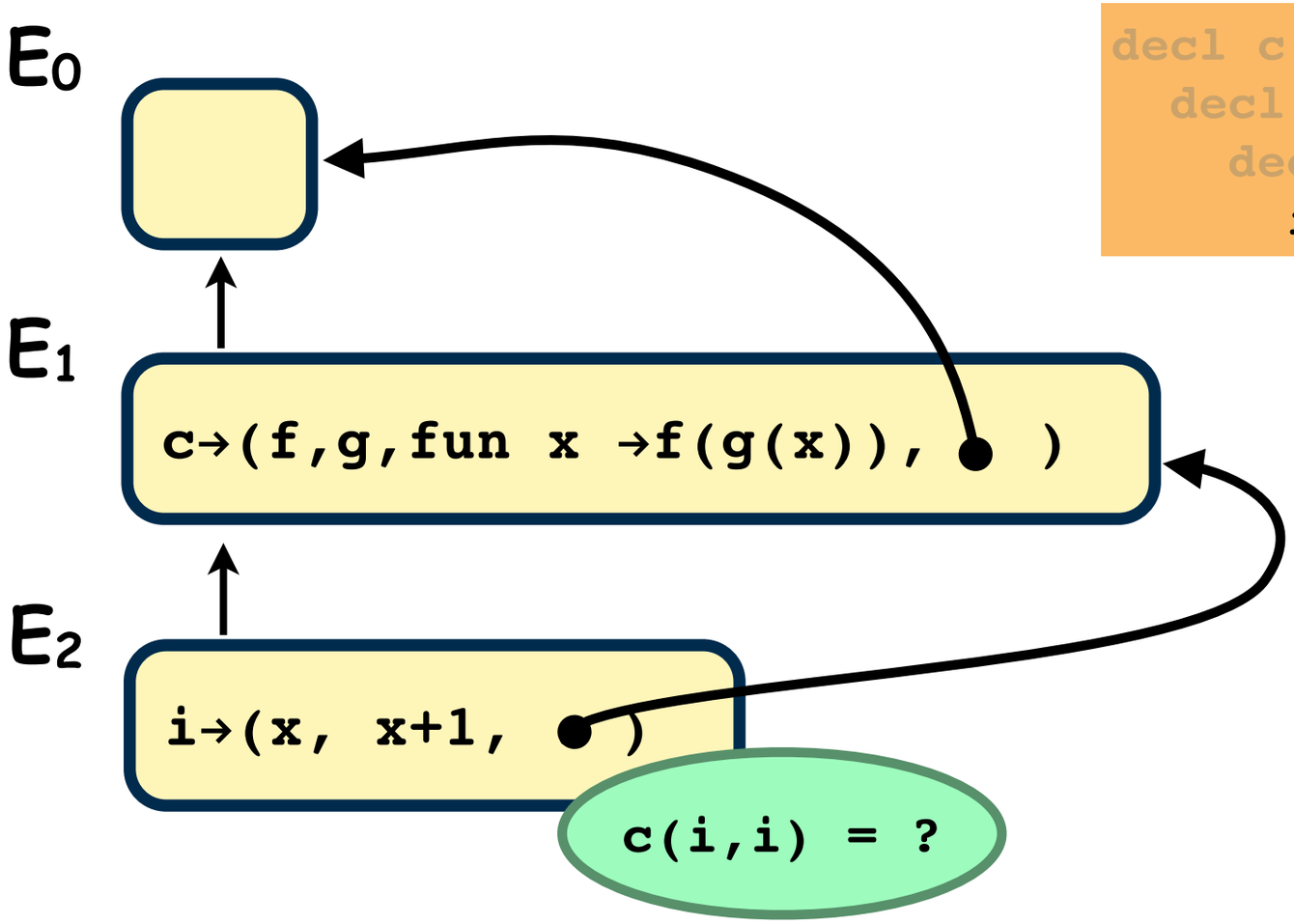


# Avaliação de Funções

```
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)
```



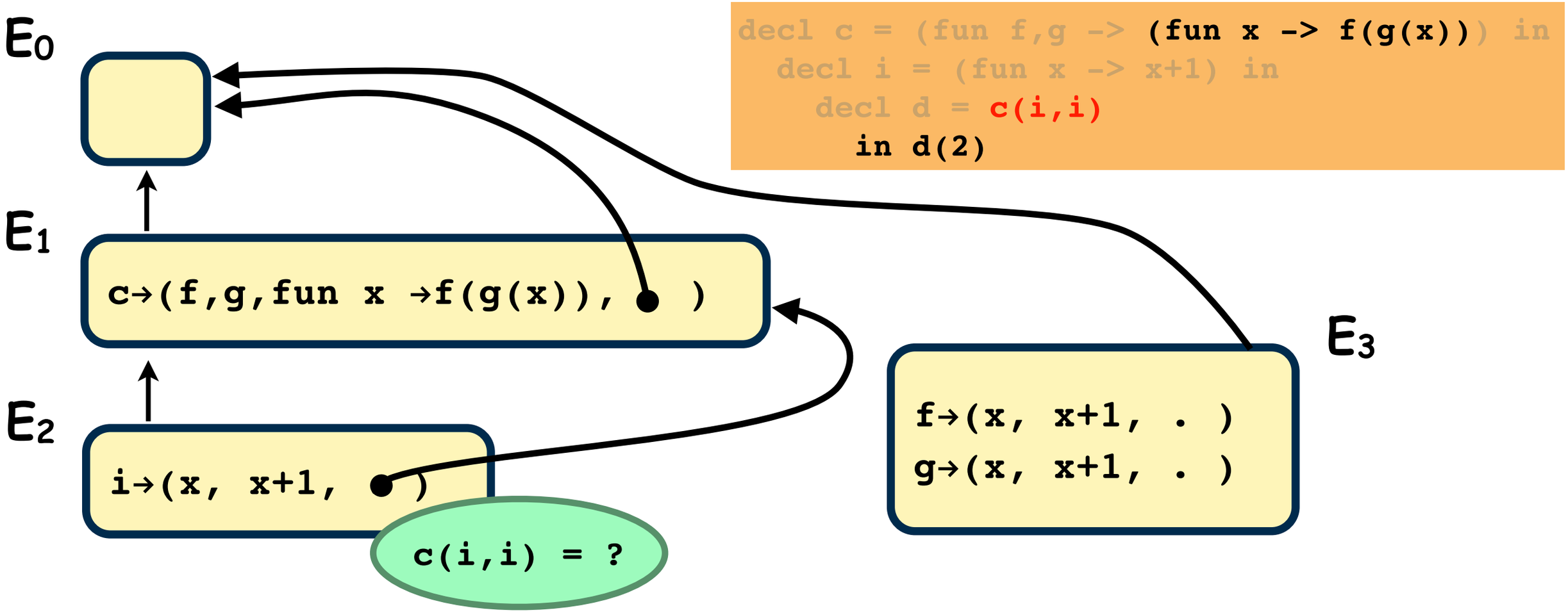
# Avaliação de Funções




```
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)
```

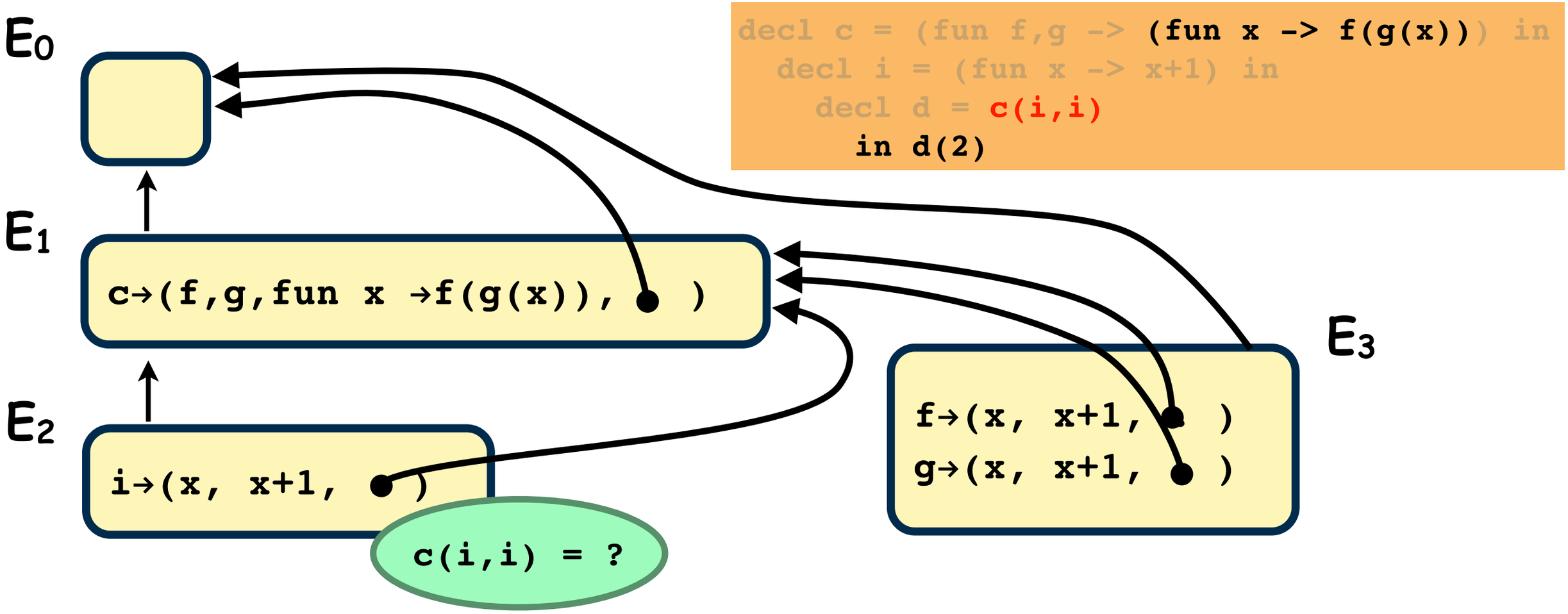
# Avaliação de Funções

```
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)
```



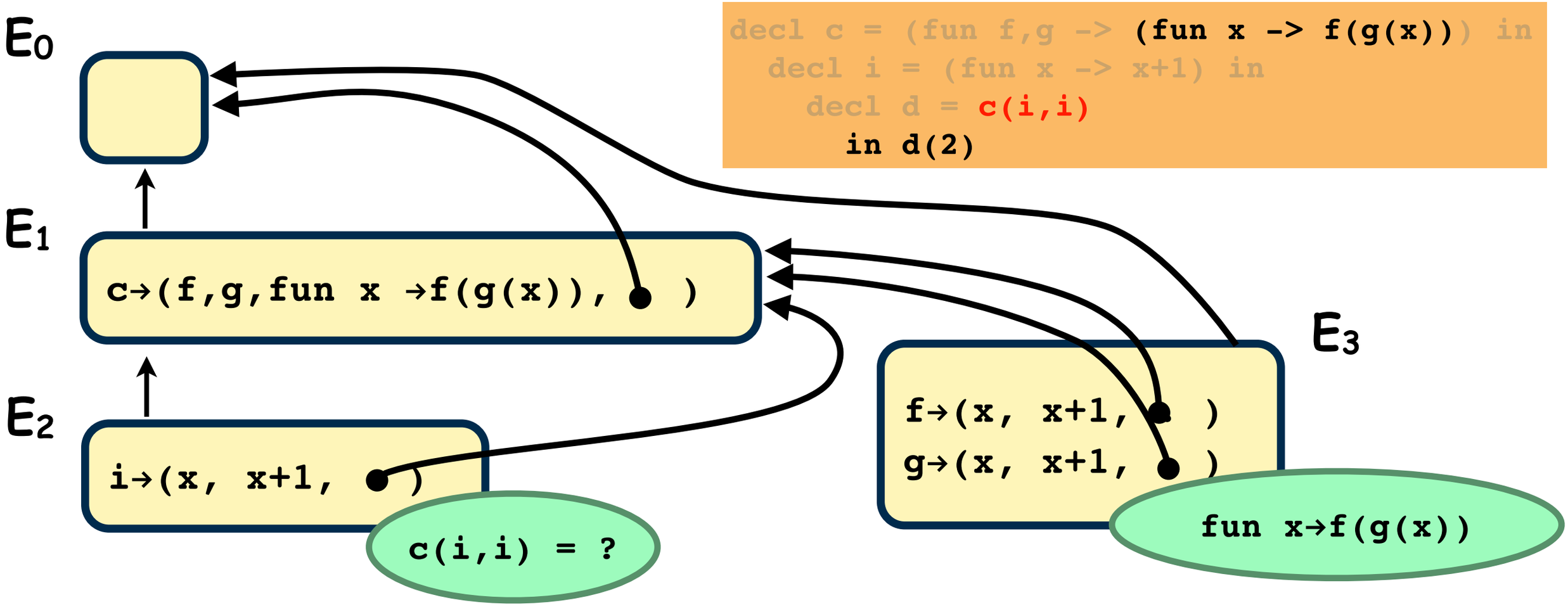
# Avaliação de Funções

```
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)
```



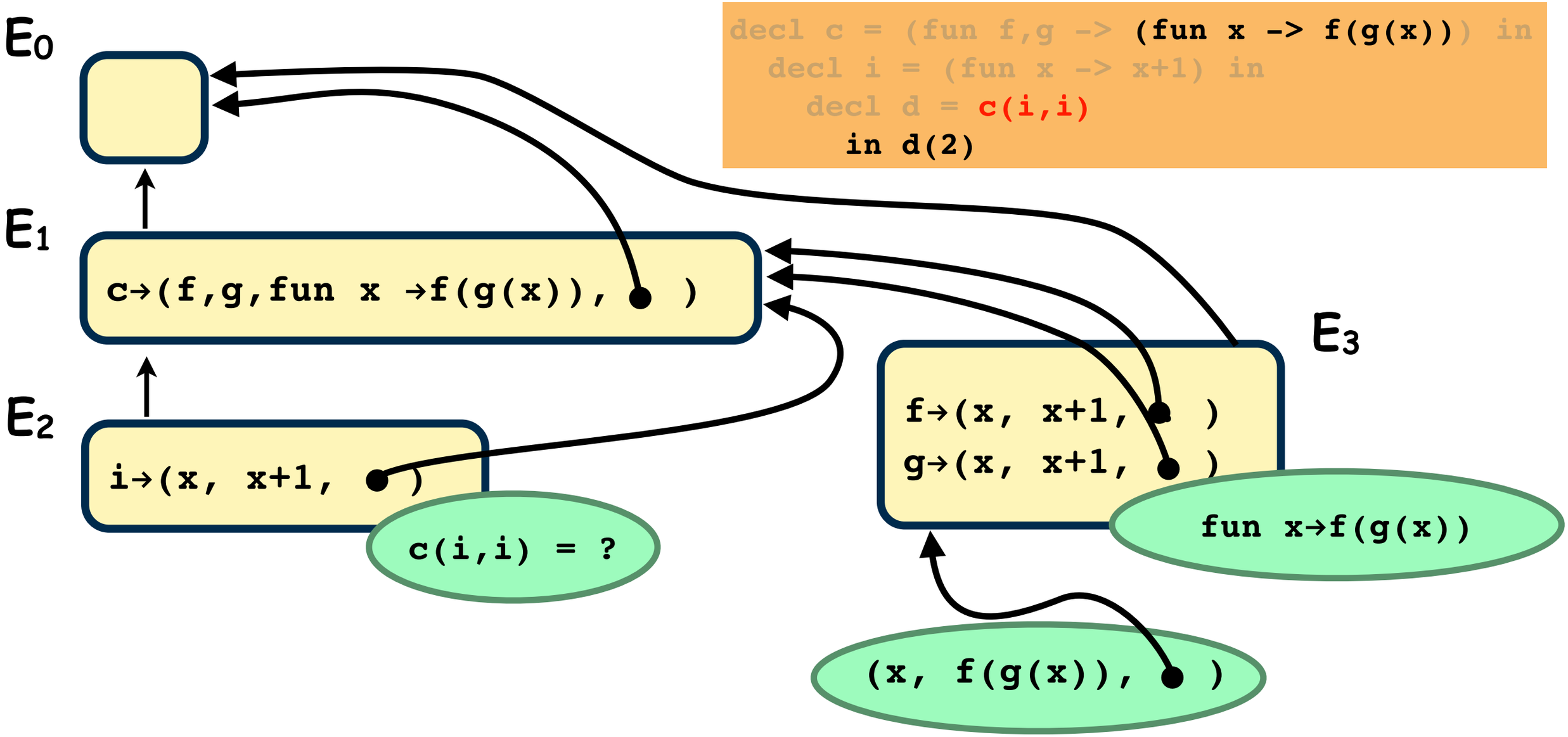
# Avaliação de Funções

```
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)
```



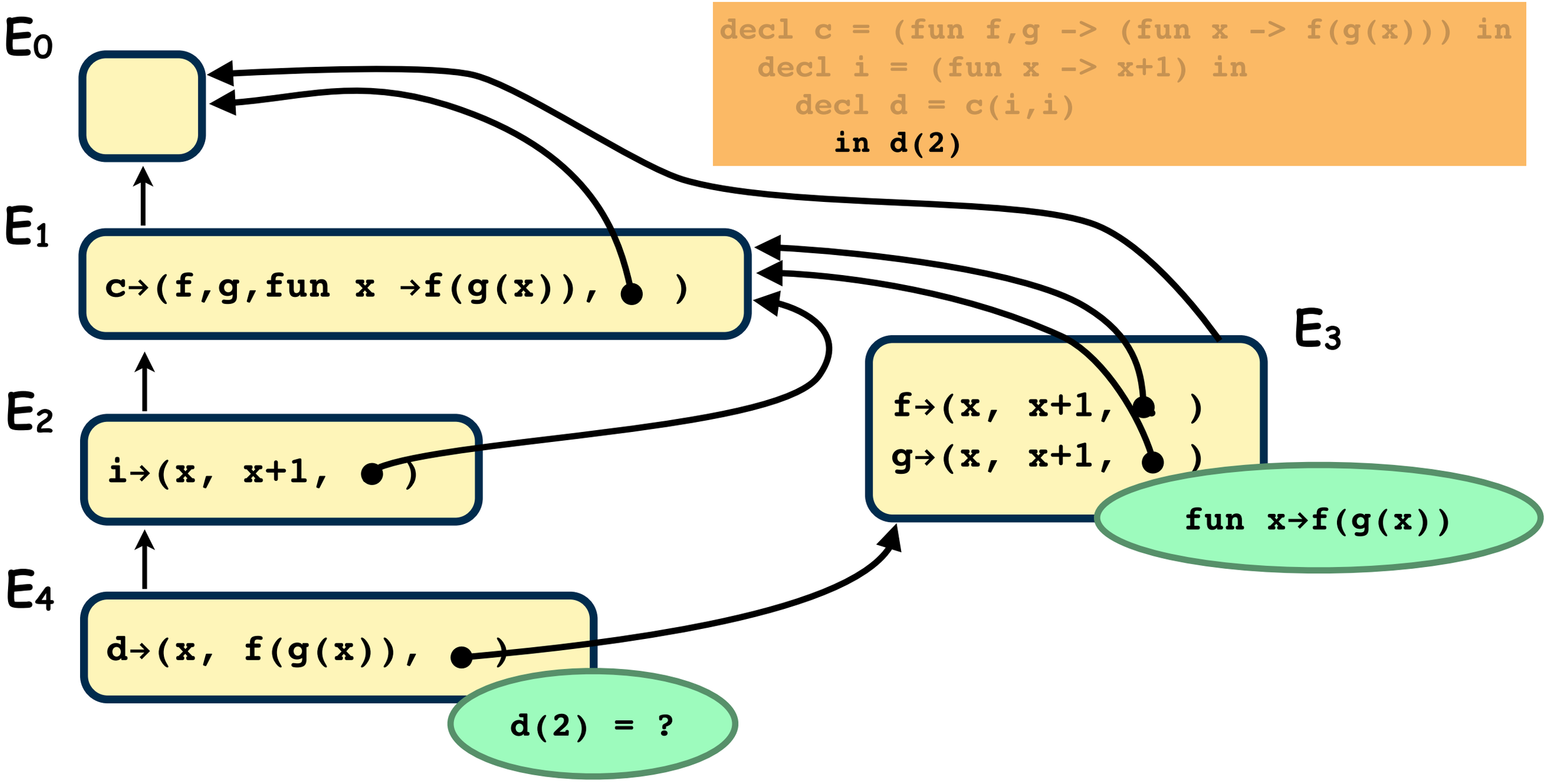
# Avaliação de Funções

```
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)
```



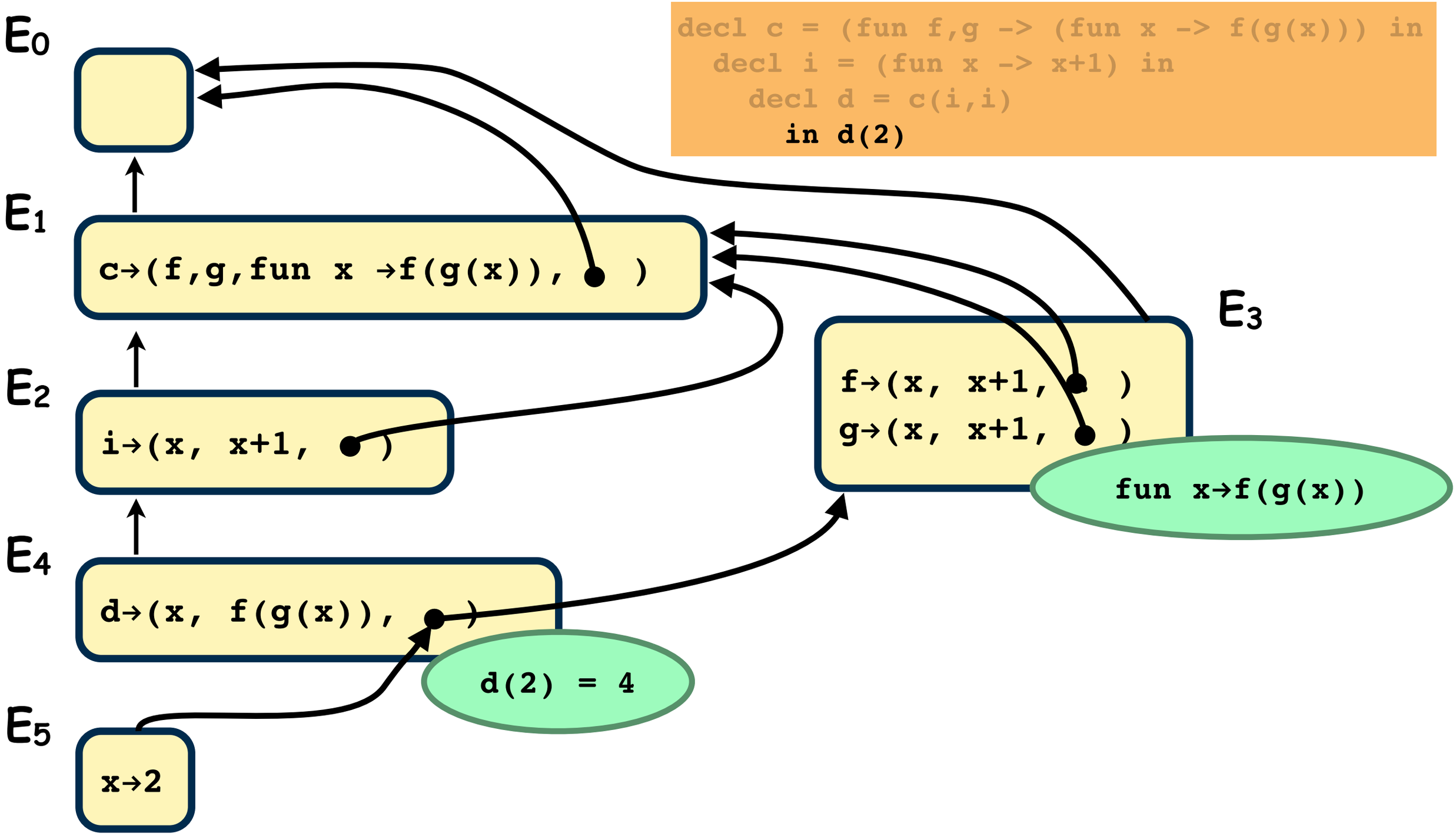
# Avaliação de Funções

```
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)
```



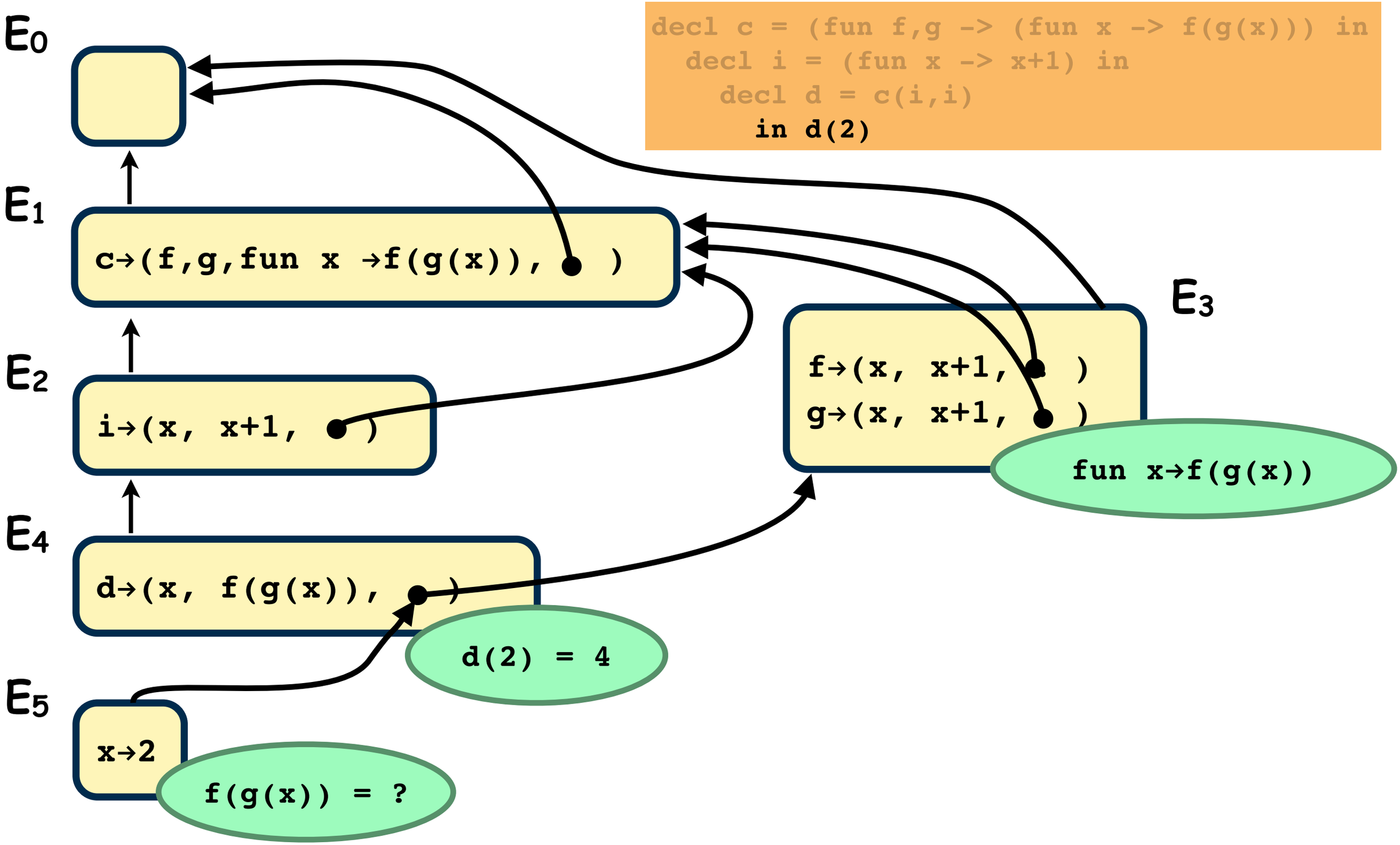
# Avaliação de Funções

```
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)
```



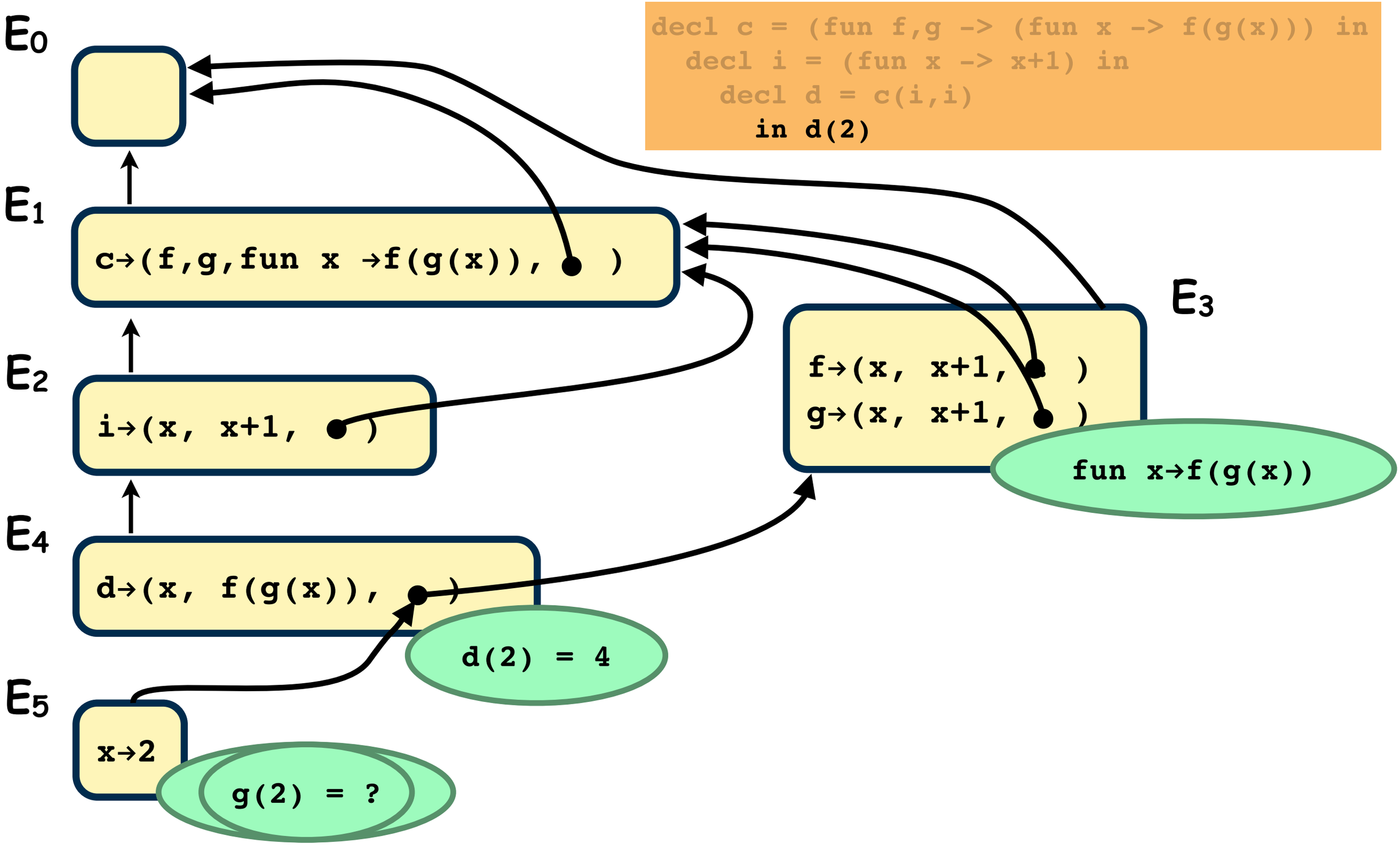
# Avaliação de Funções

```
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)
```



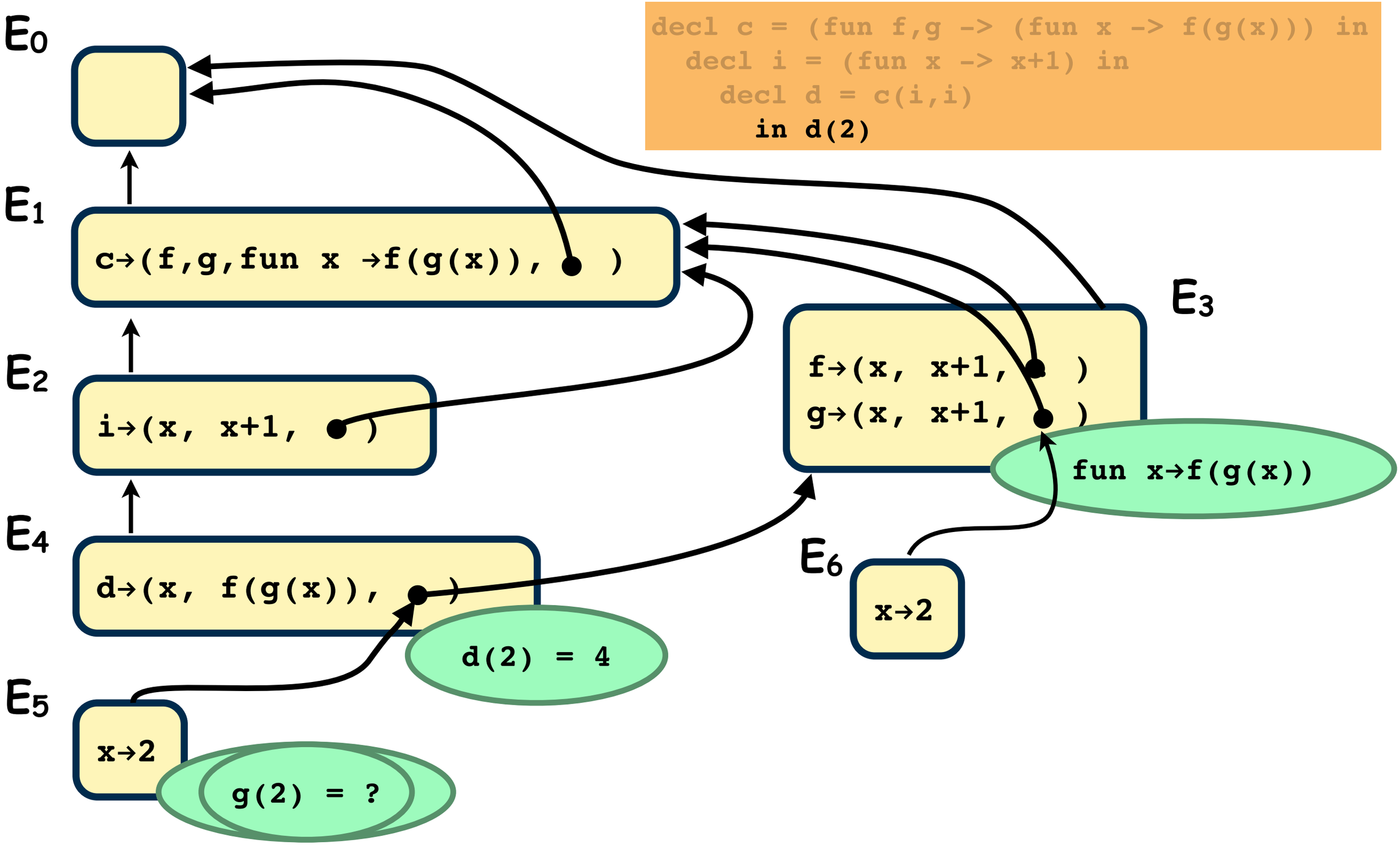
# Avaliação de Funções

```
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)
```



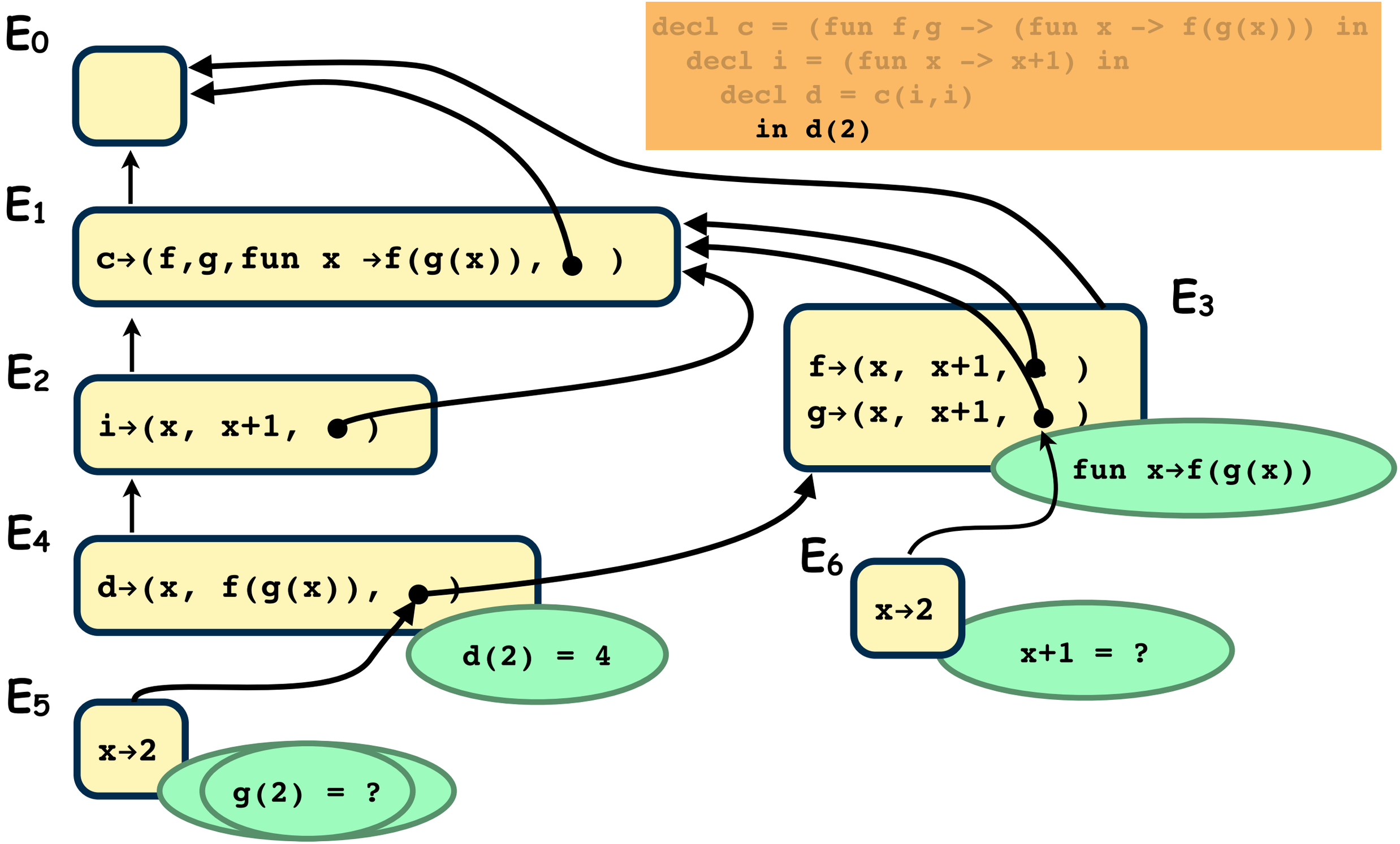
# Avaliação de Funções

```
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)
```



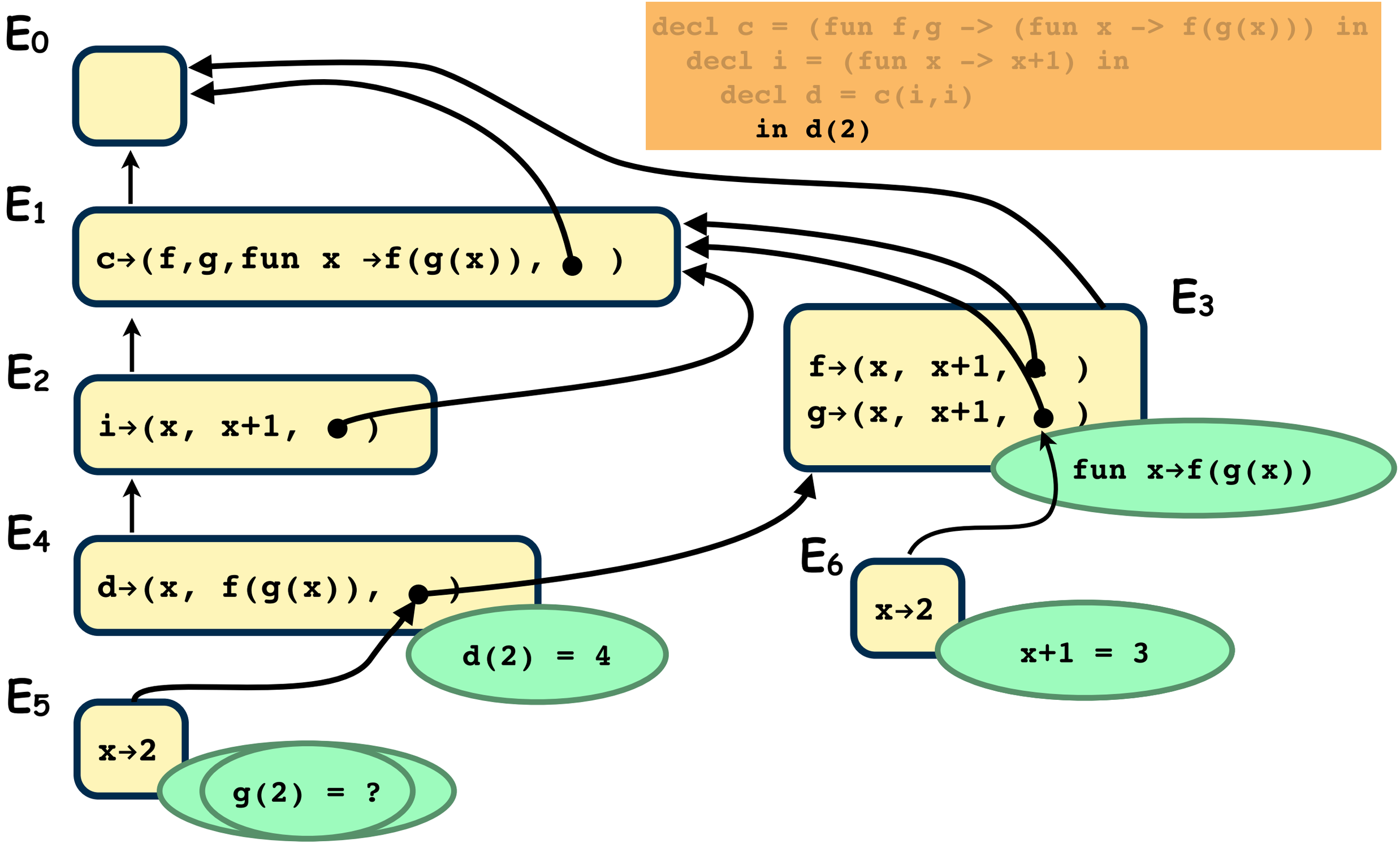
# Avaliação de Funções

```
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)
```



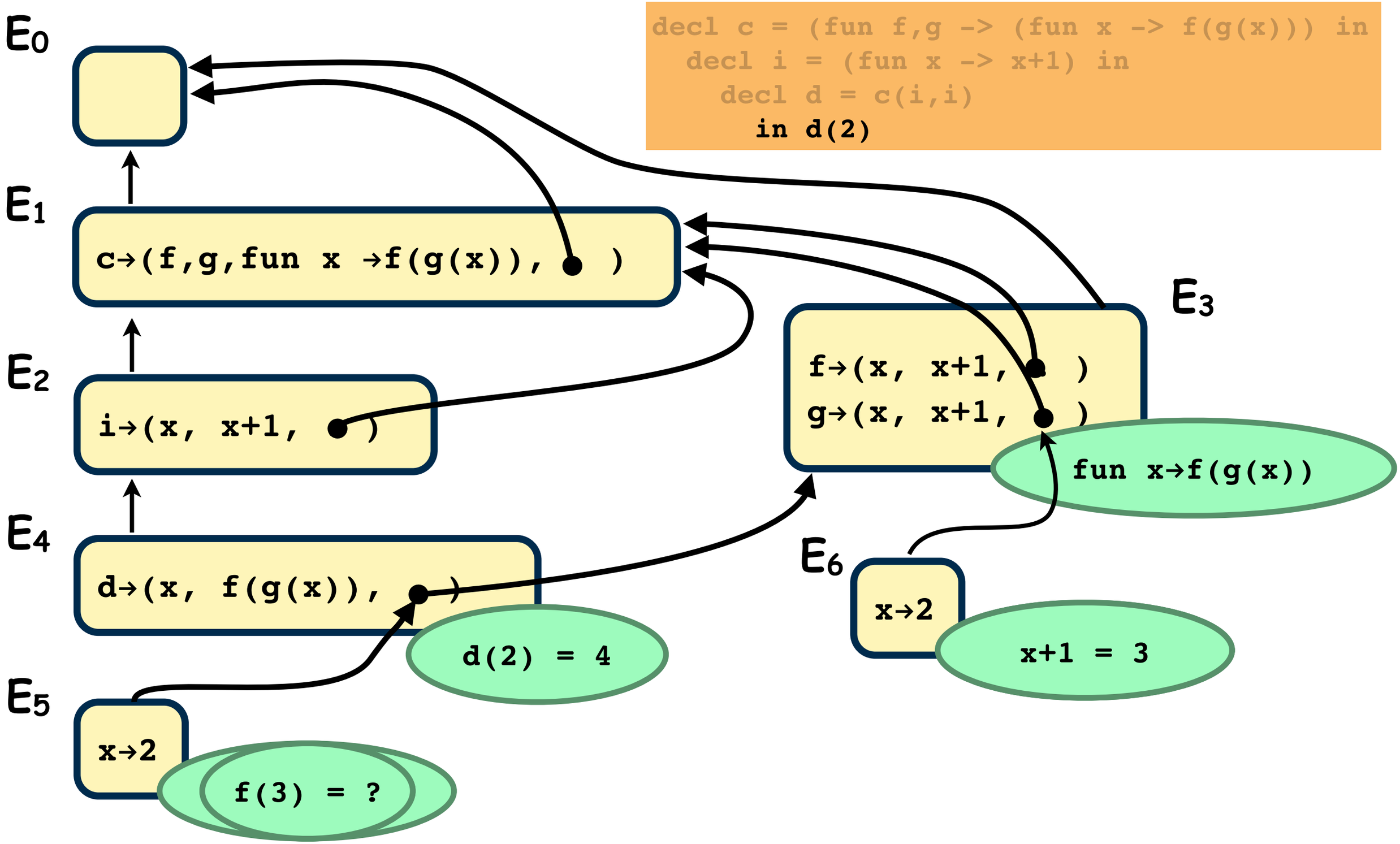
# Avaliação de Funções

```
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)
```



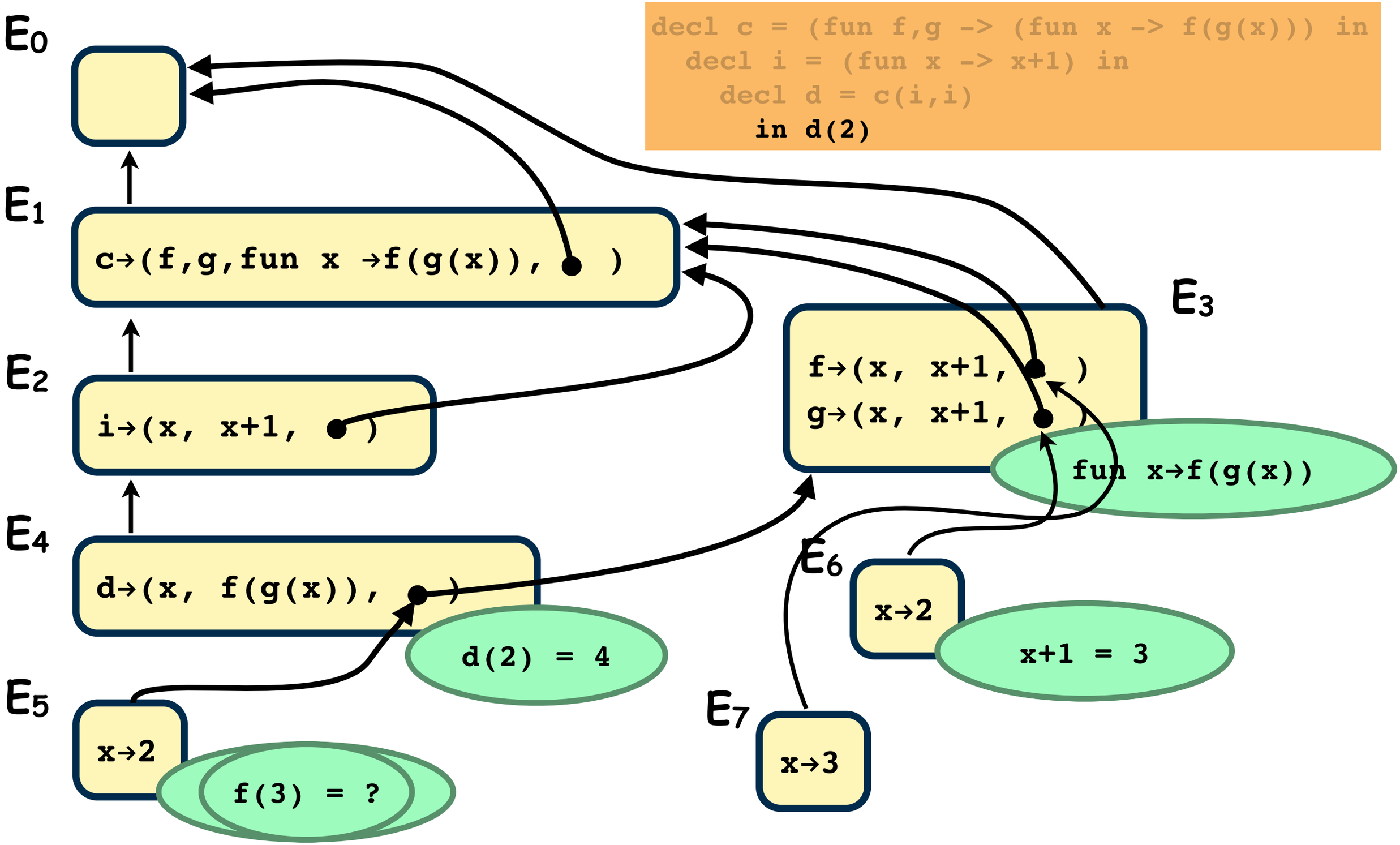
# Avaliação de Funções

```
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)
```



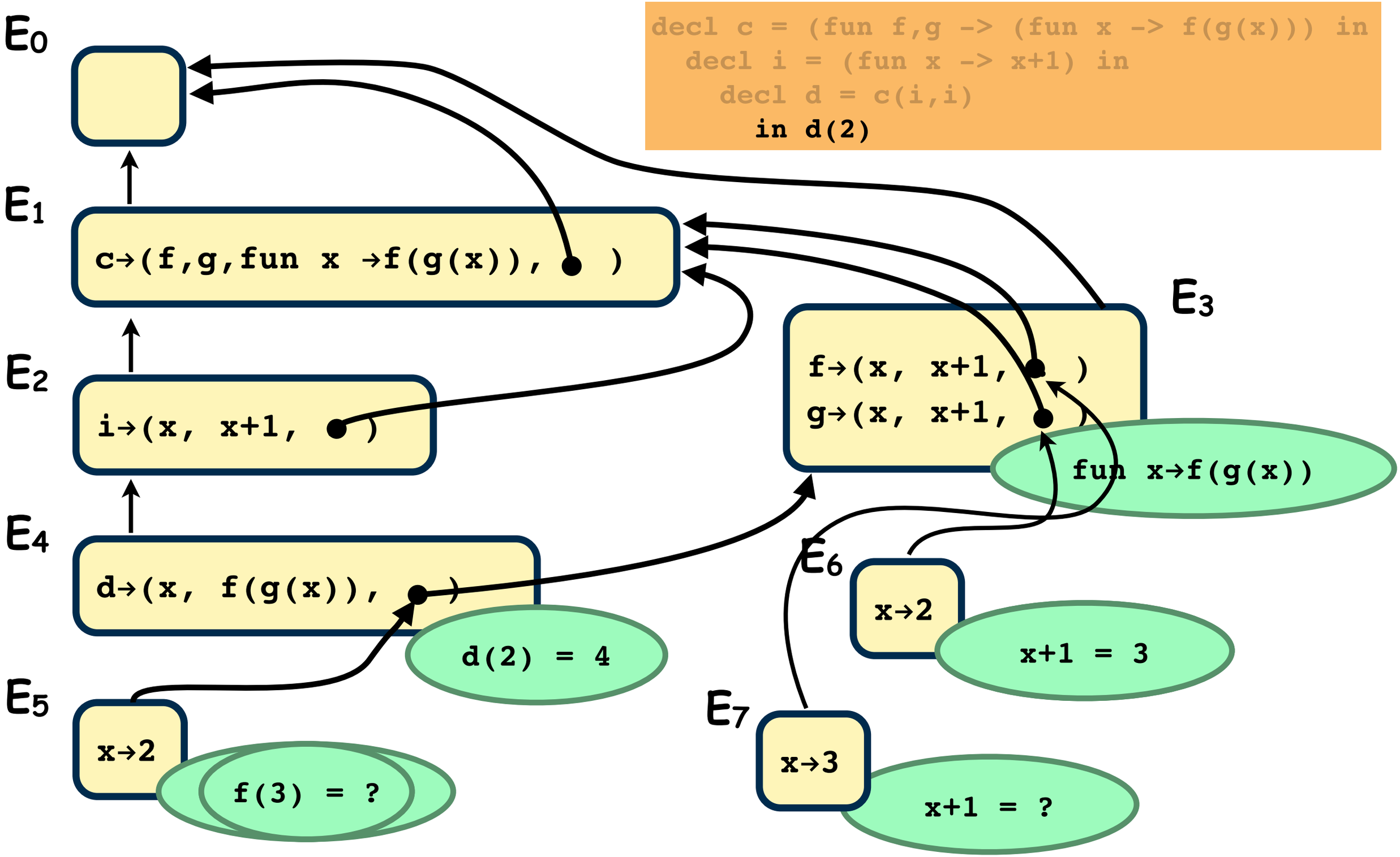
# Avaliação de Funções

```
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)
```



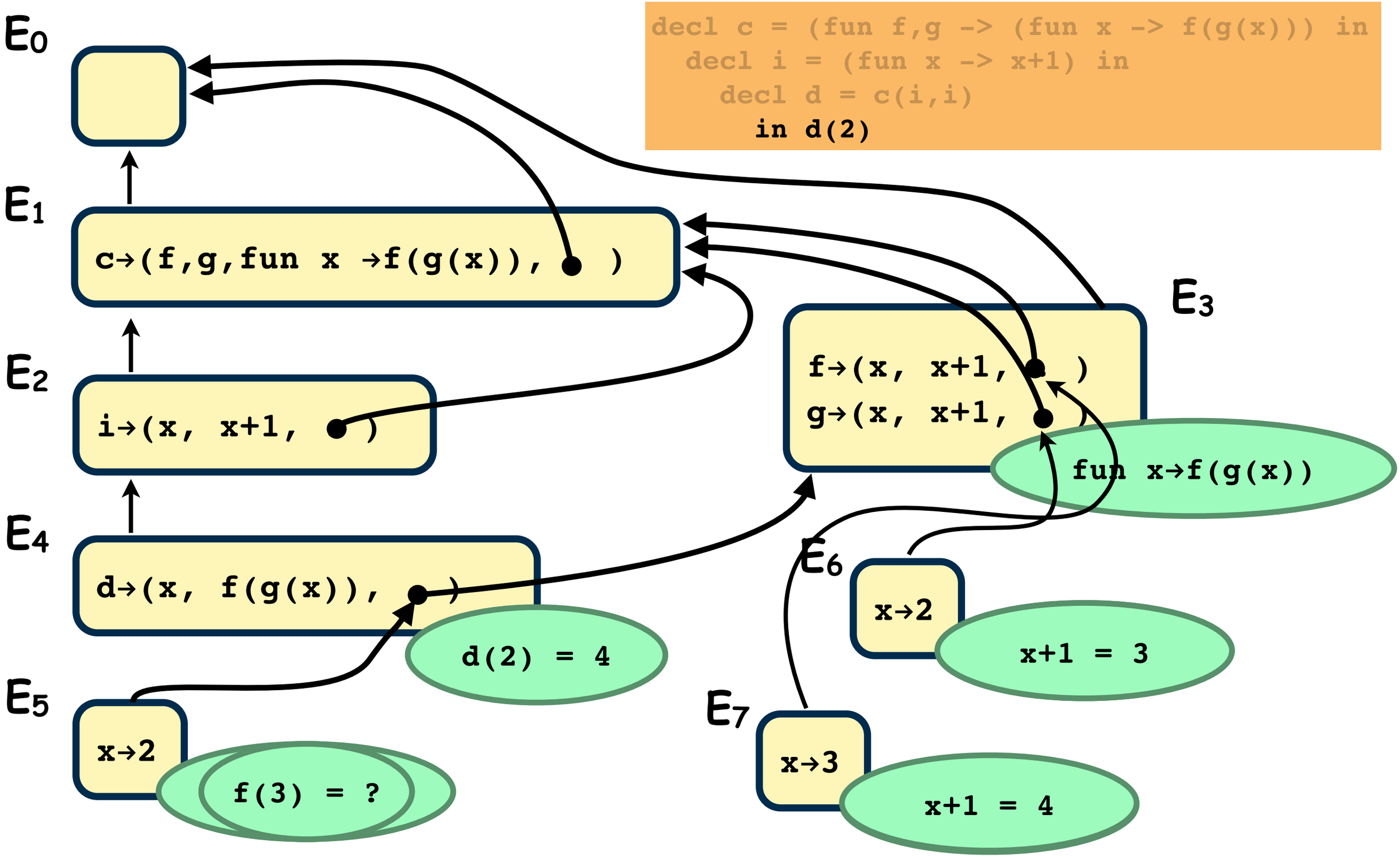
# Avaliação de Funções

```
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)
```



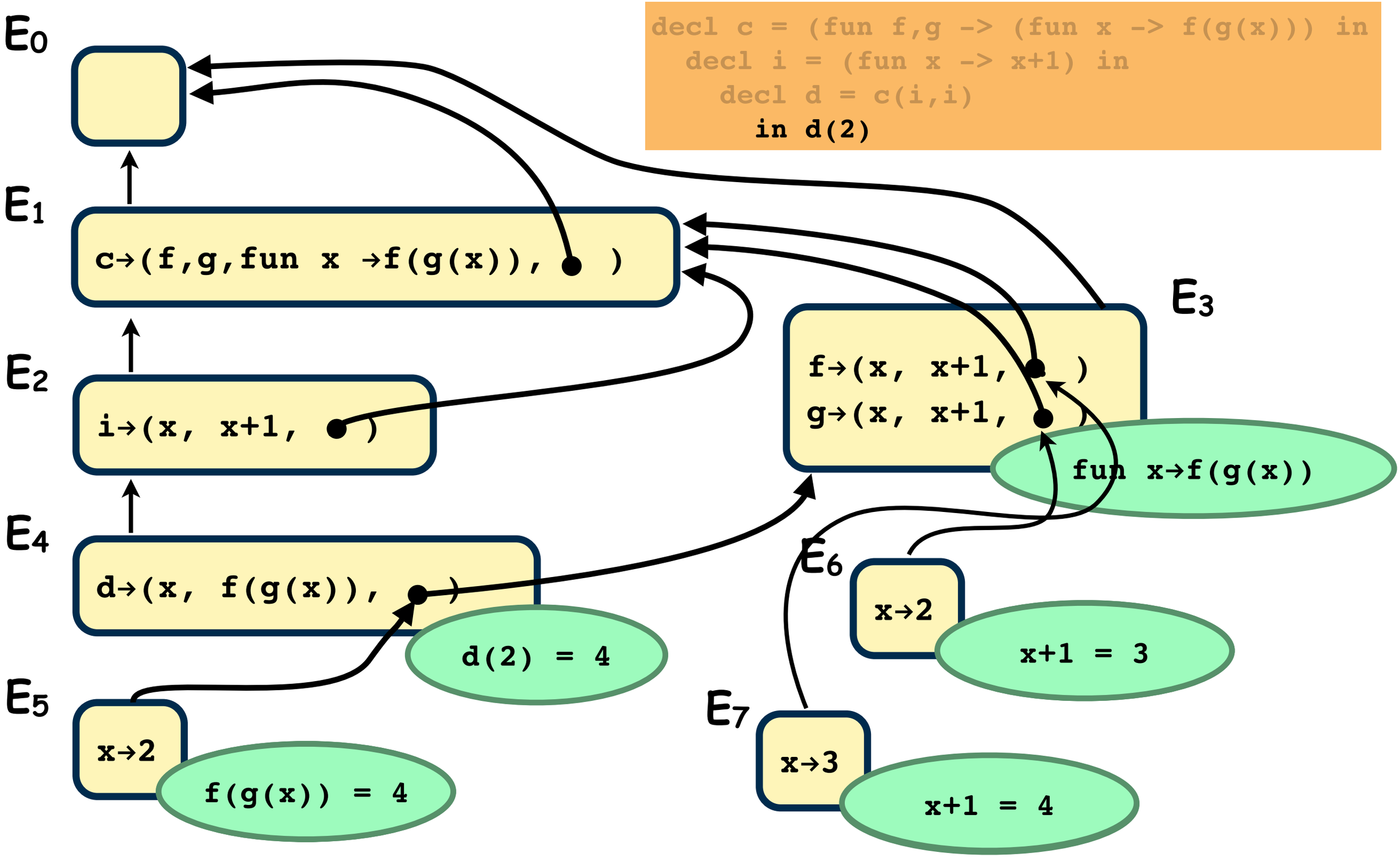
# Avaliação de Funções

```
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)
```



# Avaliação de Funções

```
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)
```



# Quiz

- Qual o valor (se existir) das seguintes expressões:

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

decl f = (fun x -> x(x)) in f(f) end
```

- Qual o valor da expressão seguinte quando avaliada pela regra dinâmica e pela regra estática de resolução de nomes:

```
decl x=2 in
decl g = (fun y -> y-x) in
decl x = 4 in g(x) end end end
```

- Considera que as duas expressões seguintes têm sempre o mesmo valor? Porquê?

```
decl id = E1 in E2 end
(fun id -> E2)( E1)
```

# Quiz (solução)

- Considera que as duas expressões seguintes têm sempre o mesmo valor? Porquê?

```
decl id = E1 in E2 end

(fun id -> E2)( E1 )
```

- Temos (aplicando as regras de avaliação):

```
eval(decl id = E1 in E2 end, env) =
  eval(E2, env.Assoc(id, eval(E1, env )))
```

- Por outro lado:

```
eval((fun id -> E2), env ) = closure(id, E2, env )

eval((fun id -> E2) (E1), env ) =
  eval(E2, env.Assoc(id, eval(E1, env )))
```

# Resumo e Leituras

- Abstração por parameterização é um mecanismo aplicado a um subprograma que o generaliza, abstraindo o valor de certas subexpressões em parâmetros bem identificados) e permite a sua instanciação e reutilização aplicada a diferentes contextos.
- Uma abstração é uma construção sintática composta por um conjunto de parâmetros e um subprograma.
- Cada instanciação de uma abstracção é activada em associação com uma interpretação dos seus parâmetros e nomes livres.
- Leituras:
  - Liskov, Guttag "Program Development in Java"
  - Friedmand "Essentials of programming languages" 3rd edition, Cap 3.
  - Mitchell, "Concepts in programming languages", Cap 7
  - Appel, Cap. 15, "Modern Compiler Implementation"

# Compilação de CALCF (1)

- A compilação de CALCF pode ser caracterizada por uma função:

$$comp: P \times ENV \rightarrow CodeSeq$$

P = Fragmento de programa (aberto)

ENV = Ambiente (função String → int )

O ambiente atribui a cada identificador uma posição na pilha de chamada (um número inteiro).

CodeSeq = Sequências de instruções

# Ingredientes...

# Chamada indirecta de funções

- A máquina CLR dispõe de instruções para a chamada de funções indirectamente a partir de valores (referências ou apontadores).

```
ldstr "Hello"
ldftn void f(object)
calli void (object)
```

- Instruções da linguagem CIL para referir e chamar funções por apontadores:
  - ldftn assinatura

    Coloca no topo da pilha um apontador para o método com a assinatura dada (completa com tipo de retorno, nome e tipos de parâmetros). ... -> ftn

    o tipo CIL dos apontadores é `native int`, em C# é `System.IntPtr`
  - calli assinatura

    Chamada indirecta de métodos. Retira do topo da pilha os argumentos e o apontador para um método a chamar. Verifica se a assinatura dada (sem nome) é compatível com o método referido pelo apontador. Deixa o resultado (se houver) no topo da pilha.

# Chamada indirecta de funções

- A estrutura de uma assembly CIL é composta por um conjunto de declarações: métodos e classes.

```
.assembly 'A' {}
.module A.dll
.class public auto ansi A extends [mscorlib]System.Object
{
  .field private int32 x
  .method ...
}
.method ... static ... object m0(object) {...}
.method ... static ... object m1(object) {
  .locals (int32)
  .entrypoint
  ...
  ret
}
```

# Chamada indirecta de funções

- A chamada indirecta de métodos estáticos passa numa primeira fase por produzir um apontador para o método (sabendo qual a assinatura completa) e numa segunda fase chamar o método tendo o apontador e sabendo a sua assinatura (sem nome).

```
.method static public void m0(object) {
  .locals init (native int)
  ldftn void m1(object)
  stloc.0
  ldstr "Hello"
  ldloc.0
  calli void(object)
  ret
}
.method static public void m1(object) {
  ldarg.0
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```

# Compilação de CALCF

Um programa CALCF contém abstracções anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral**: Associar a cada abstracção CALCF um método estático em CIL que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
 ...
}
```

# Compilação de CALCF

Um programa CALCF contém abstracções anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral**: Associar a cada abstracção CALCF um método estático em CIL que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
 ...
}
.method ... static ... object f0(...)
{...

.method ... static ... object f1(...)
{...

.method ... static ... object f2(...)
{...

.method ... static ... object f3(...)
{...
```

# Compilação de CALCF

Um programa CALCF contém abstracções anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral**: Associar a cada abstracção CALCF um método estático em CIL que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
 ...
}
.method ... static ... object f0(...)
{... [[ y + x ]]

.method ... static ... object f1(...)
{...

.method ... static ... object f2(...)
{...

.method ... static ... object f3(...)
{...
```

# Compilação de CALCF

Um programa CALCF contém abstracções anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral**: Associar a cada abstracção CALCF um método estático em CIL que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
 ...
}
.method ... static ... object f0(...)
{...

.method ... static ... object f1(...)
{... [[ f(x)+1 ]]

.method ... static ... object f2(...)
{...

.method ... static ... object f3(...)
{...
```

# Compilação de CALCF

Um programa CALCF contém abstracções anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral**: Associar a cada abstracção CALCF um método estático em CIL que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
 ...
}
.method ... static ... object f0(...)
{...

.method ... static ... object f1(...)
{...

.method ... static ... object f2(...)
{... [[ (fun x -> x)(g(y)) ]]

.method ... static ... object f3(...)
{...
```

# Compilação de CALCF

Um programa CALCF contém abstracções anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral**: Associar a cada abstracção CALCF um método estático em CIL que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
 ...
}
.method ... static ... object f0(...)
{...

.method ... static ... object f1(...)
{...

.method ... static ... object f2(...)
{...

.method ... static ... object f3(...)
{... [[ x ]]
```

# Compilação de CALCF

- A compilação de CALCF pode ser caracterizada por uma função:

$$comp: P \times ENV \rightarrow CodeSeq \times CodeSeq\ list$$

P = Fragmento de programa (aberto)

ENV = Ambiente (função String → int)

O ambiente atribui a cada identificador uma posição na pilha de chamada (um número inteiro).

CodeSeq = Sequências de instruções

# Compilação de CALCF

No processo de compilação das linguagens anteriores, os valores (constantes) associados aos identificadores são guardados em vectores locais ao método CIL que implementa um programa. Ao espalhar a compilação de um programa por vários métodos CIL perde-se o acesso aos identificadores que não estão no ambiente mais próximo. A linguagem CIL não suporta o aninhamento de funções da linguagem CALCF, i.e. as variáveis locais CIL são apenas acessíveis dentro do método.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
  .locals (object[] table)
  ...
}
.method ... static ... object f0(object y)
{
  ldloc table // x ????
  ...
}
.method ... static ... object f1(object x)
{
  ldloc table // g ????
  ...
}
...
```

# Resolução estática de nomes...

# ... em Linguagens tipo Algol

- As linguagens tipo Algol suportam aninhamento (nesting) de declarações e o acesso a variáveis não locais é feito através de ligações entre registos na pilha de execução (static link).

```
Program P;
var x:integer;

Function f0(y:integer):integer;
  var z:integer;

  Function f1(w:integer):integer;
  begin
   f1 := w+y+x+z
  end

  Function f2(w:integer):integer;
  begin
    f2 := f3(f1)
  end

begin
  f0 := f2(f1(y))
end
```

```
Function f3(function
f(y:integer):integer):integer;
  begin
    f3 := f(x)+x
  end

Function f4(x:integer):integer;
  begin
    f4 := x+1
  end


begin
  x := 1;
  f4(f0(1))
end.
```

# ... em Linguagens tipo Algol

- Acesso a identificadores compilado para acesso a endereços de memória na stack.
- Cada chamada de uma função ou procedimento gera um bloco na pilha com informação relavante:
  - Endereço de retorno
  - Link dinâmico (bloco da função que o chamou)
  - Link estático (bloco envolvente no programa)
  - Memória para parâmetros/argumentos
  - Memória para variáveis locais
- Os blocos são desempilhados quando a função retorna.

```
Program P;
var x:integer;

Function f0(y:integer):Integer;
  var z:integer;

  Function f1(w:integer):Integer;
    begin
     f1 := w+y+x+z
    end

  Function f2(w:integer):Integer;
    begin
      f2 := f3(f1)
    end

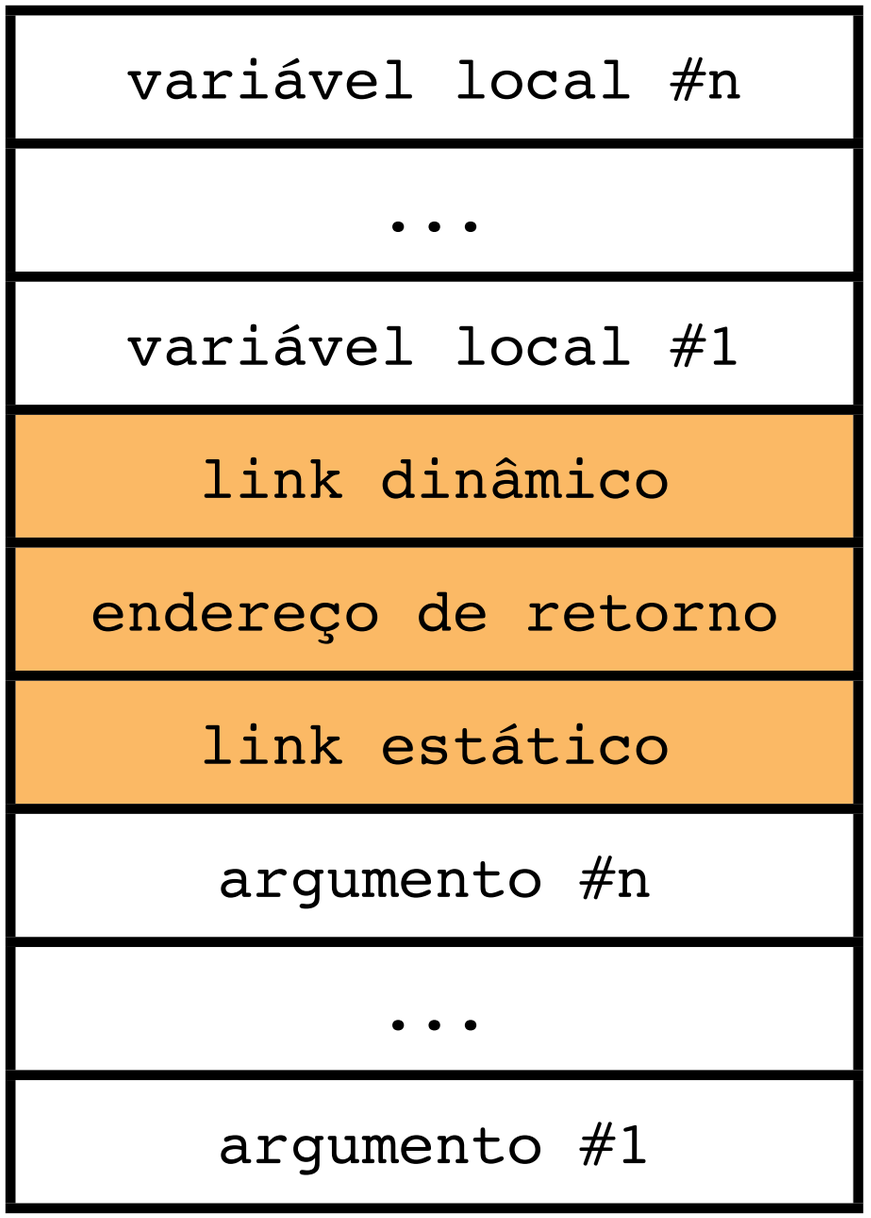
  begin
    f0 := f2(y)
  end

Function f3(function f(y:integer):integer):Integer;
  begin
    f3 := f(x)+x
  end

Function f4(x:integer):Integer;
   ...

begin
  x := 1;
  f0(1)
end.
```

## Stack frame



```
Program P;
var x:integer;

Function f0(y:integer):Integer;
  var z:integer;

  Function f1(w:integer):Integer;
    begin
     f1 := w+y+x+z
    end

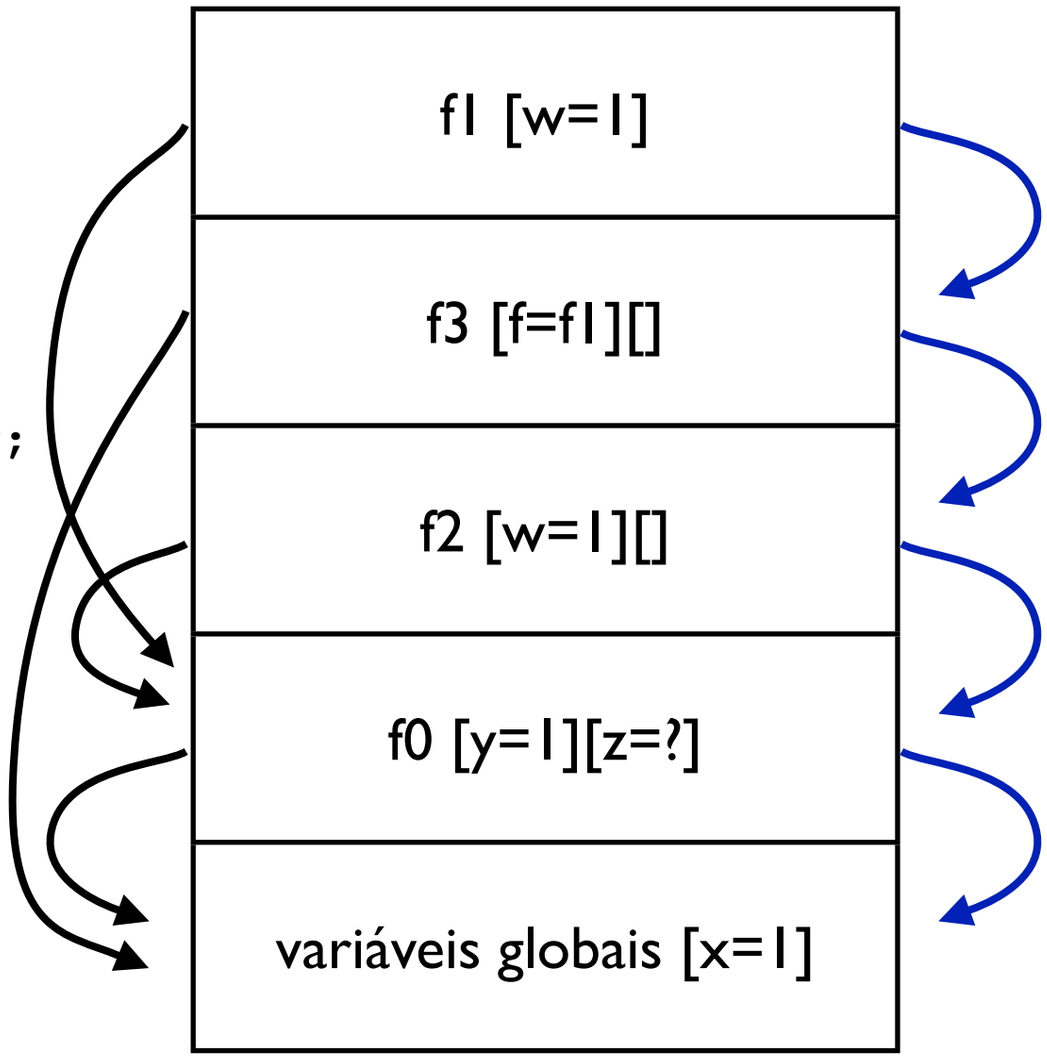
  Function f2(w:integer):Integer;
    begin
      f2 := f3(f1)
    end

  begin
    f0 := f2(y)
  end

Function f3(function f(y:integer):integer):Integer;
  begin
    f3 := f(x)+x
  end

Function f4(x:integer):Integer;
   ...

begin
  x := 1;
  f0(1)
end.
```



```
Program P;
var x:integer;

Function f0(y:integer):Integer;
  var z:integer;

  Function f1(w:integer):Integer;
    begin
     f1 := w+y+x+z
    end

  Function f2(w:integer):Integer;
    begin
      f2 := f3(f1)
    end

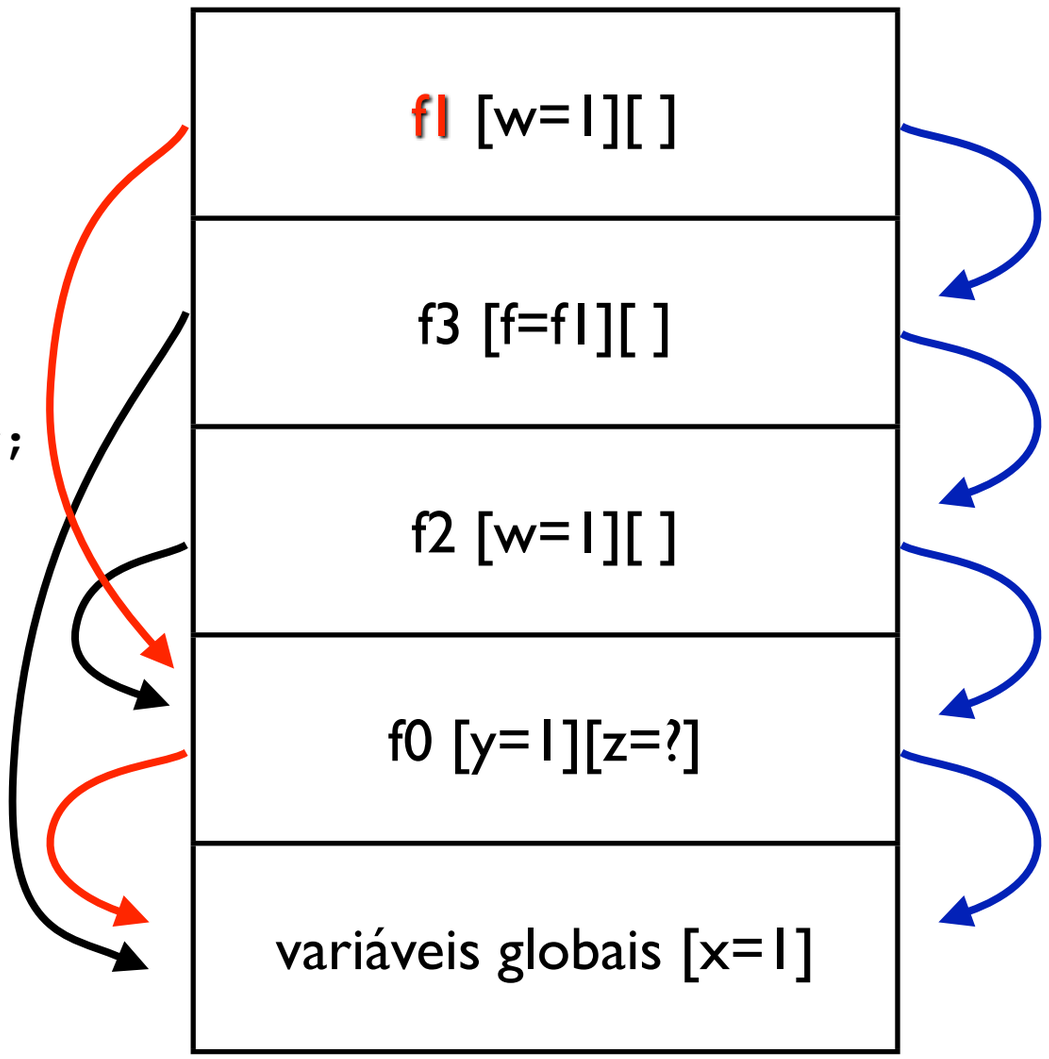
  begin
    f0 := f2(y)
  end

Function f3(function f(y:integer):integer):Integer;
  begin
    f3 := f(x)+x
  end

Function f4(x:integer):Integer;
   ...

begin
  x := 1;
  f0(1)
end.
```

| id | (saltos,offset) |
|---|---|
| w | (0,p#1) |
| y | (1,p#1) |
| x | (2,v#1) |
| z | (1,v#1) |



```
Program P;
var x:integer;

Function f0(y:integer):Integer;
  var z:integer;

  Function f1(w:integer):Integer;
    begin
     f1 := w+y+x+z
    end

  Function f2(w:integer):Integer;
    begin
      f2 := f3(f1)
    end

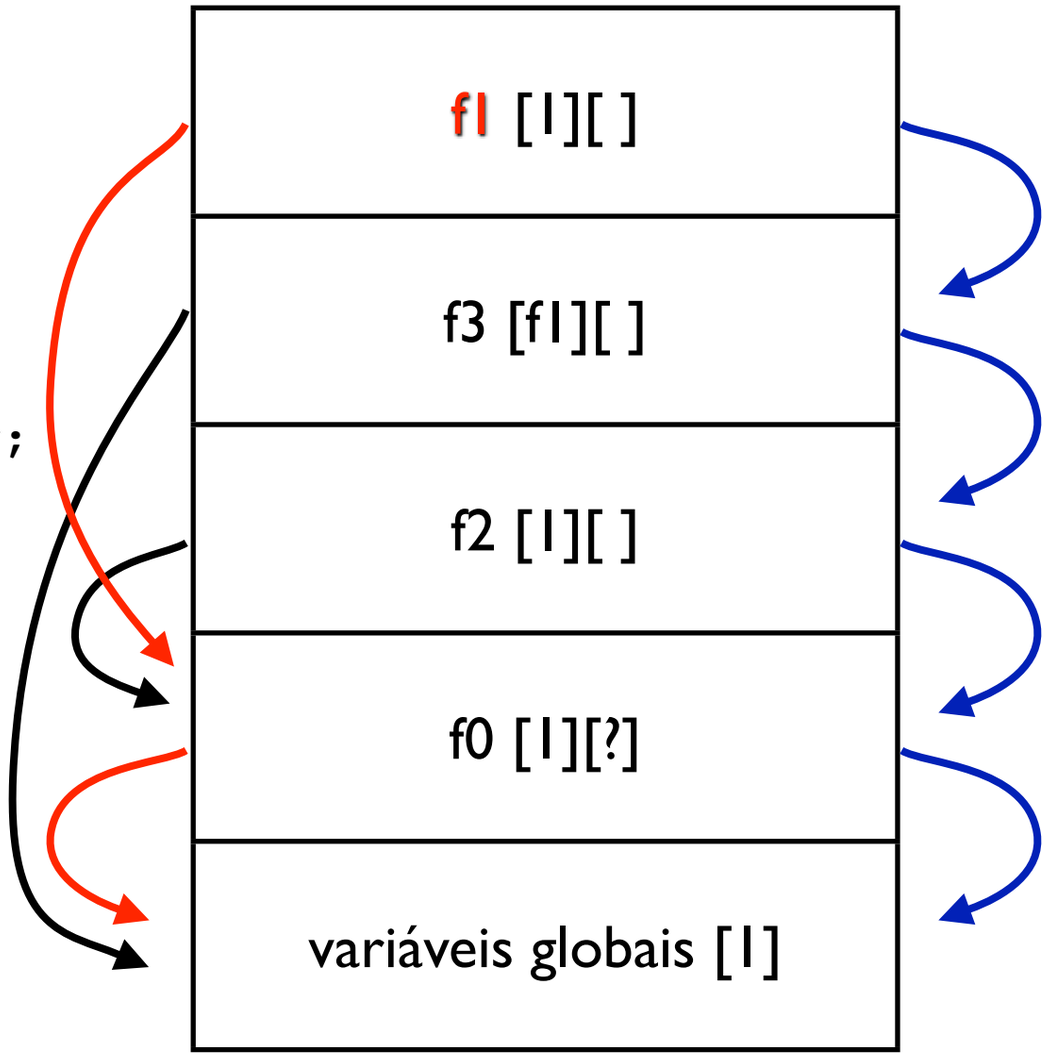
  begin
    f0 := f2(y)
  end

Function f3(function f(y:integer):integer):Integer;
  begin
    f3 := f(x)+x
  end

Function f4(x:integer):Integer;
   ...

begin
  x := 1;
  f0(1)
end.
```

| id | (saltos,offset) |
|---|---|
| w | (0,p#1) |
| y | (1,p#1) |
| x | (2,v#1) |
| z | (1,v#1) |



# ... em Linguagens tipo C

- Acesso a identificadores compilado para acesso a endereços de memória na stack.
- Cada chamada de uma função ou procedimento gera um bloco na pilha com informação relavante:
  - Endereço de retorno
  - Link dinâmico (bloco da função que o chamou)
  - Memória para parâmetros/argumentos
  - Memória para variáveis locais
- Não é necessário nenhum link estático. [Porquê?]

# ... em Linguagens tipo ML

- Acesso a identificadores compilado para acesso a endereços de memória na stack.
- Cada chamada de uma função ou procedimento gera um bloco na pilha com informação relavante
  - Memória para parâmetros/argumentos
  - Memória para variáveis locais
  - [Que mais?]
- Não é suficiente o link estático numa pilha simples. [Porquê?]

Funções de ordem superior... High order functions

# ... em Linguagens tipo ML

- Acesso a identificadores compilado para acesso a endereços de memória na stack.
- Cada chamada de uma função ou procedimento gera um bloco na pilha com informação relavante
  - Memória para parâmetros/argumentos
  - Memória para variáveis locais
  - [Que mais?]
- Não é suficiente o link estático numa pilha simples.

Funções de ordem superior... High order functions

Porque as funções são valores de runtime (closures) e os stack frames não podem ser desempilhados.

# Compilação de CALCF

- A compilação de CALCF pode ser caracterizada por uma função:

$$comp: P \times ENV \rightarrow CodeSeq \times CodeSeq\ list$$

P = Fragmento de programa (aberto)

ENV = Ambiente (função String → int)

O ambiente atribui a cada identificador uma posição na pilha de chamada (um número inteiro).

CodeSeq = Sequências de instruções

# Compilação de CALCF (Ideias Gerais)

- Cada abstracção é mapeada numa função CIL com um registo de activação próprio onde são guardados os valores correspondentes aos argumentos e às declarações locais.

- O registo de activação contém um link estático para o registo de activação da última ocorrência na pilha da abstracção envolvente estaticamente no programa. Posição 0 do registo de activação.

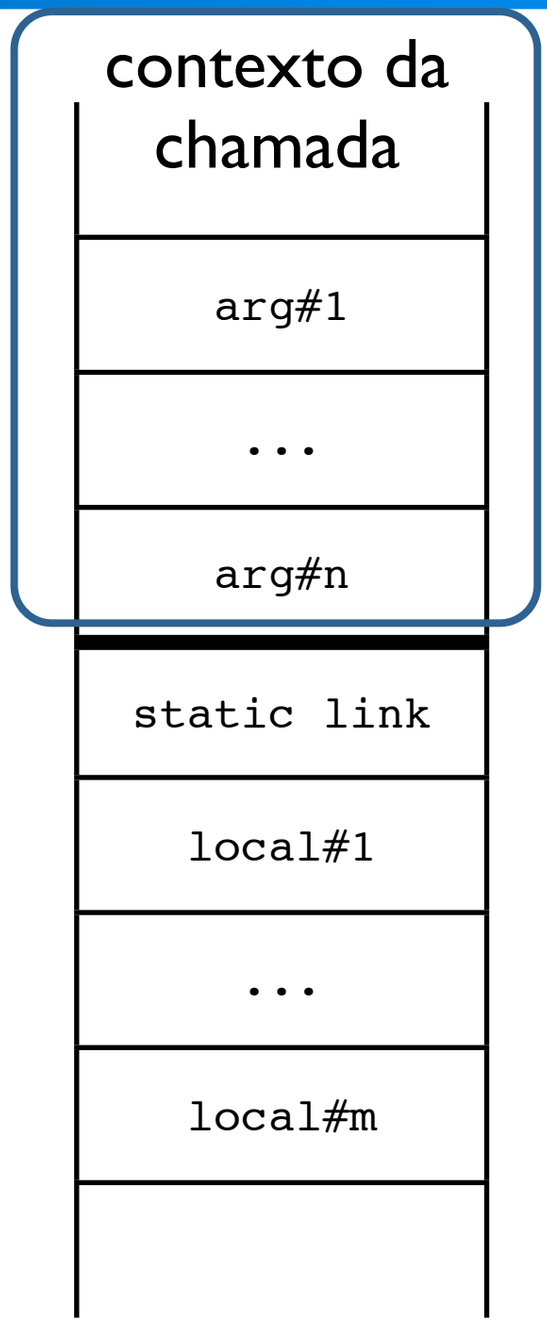
- Os argumentos são passados através do registo de activação, que é passado pelo mecanismo normal da linguagem CIL. Os valores são colocados no registo de activação **antes** da chamada. Numa pilha fazem parte do registo de activação anterior e têm deslocamentos negativos em relação ao FramePointer.

# Compilação de CALCF (Ideias Gerais)

- Cada abstracção é mapeada numa função CIL com um registo de activação próprio onde são guardados os valores correspondentes aos argumentos e às declarações locais.

- O registo de activação contém um link estático para o registo de activação da última ocorrência na pilha da abstracção envolvente estaticamente no programa. Posição 0 do registo de activação.

- Os argumentos são passados através do registo de activação, que é passado pelo mecanismo normal da linguagem CIL. Os valores são colocados no registo de activação **antes** da chamada. Numa pilha fazem parte do registo de activação anterior e têm deslocamentos negativos em relação ao FramePointer.

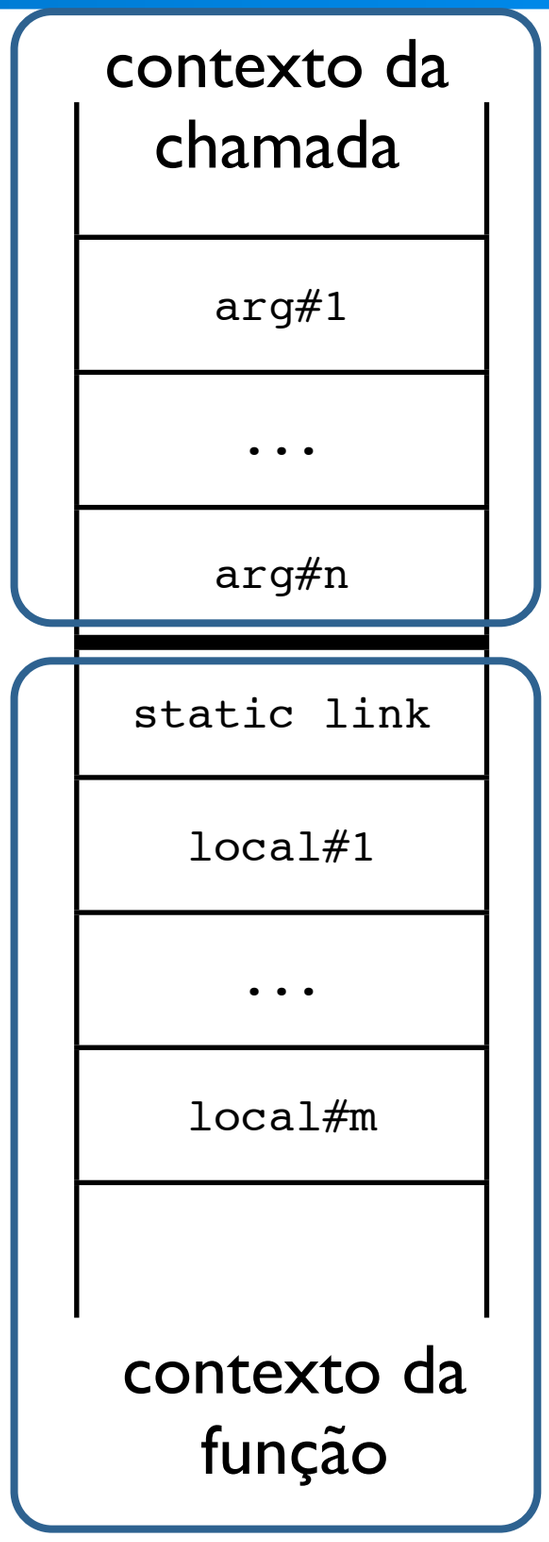


# Compilação de CALCF (Ideias Gerais)

- Cada abstracção é mapeada numa função CIL com um registo de activação próprio onde são guardados os valores correspondentes aos argumentos e às declarações locais.

- O registo de activação contém um link estático para o registo de activação da última ocorrência na pilha da abstracção envolvente estaticamente no programa. Posição 0 do registo de activação.

- Os argumentos são passados através do registo de activação, que é passado pelo mecanismo normal da linguagem CIL. Os valores são colocados no registo de activação **antes** da chamada. Numa pilha fazem parte do registo de activação anterior e têm deslocamentos negativos em relação ao FramePointer.

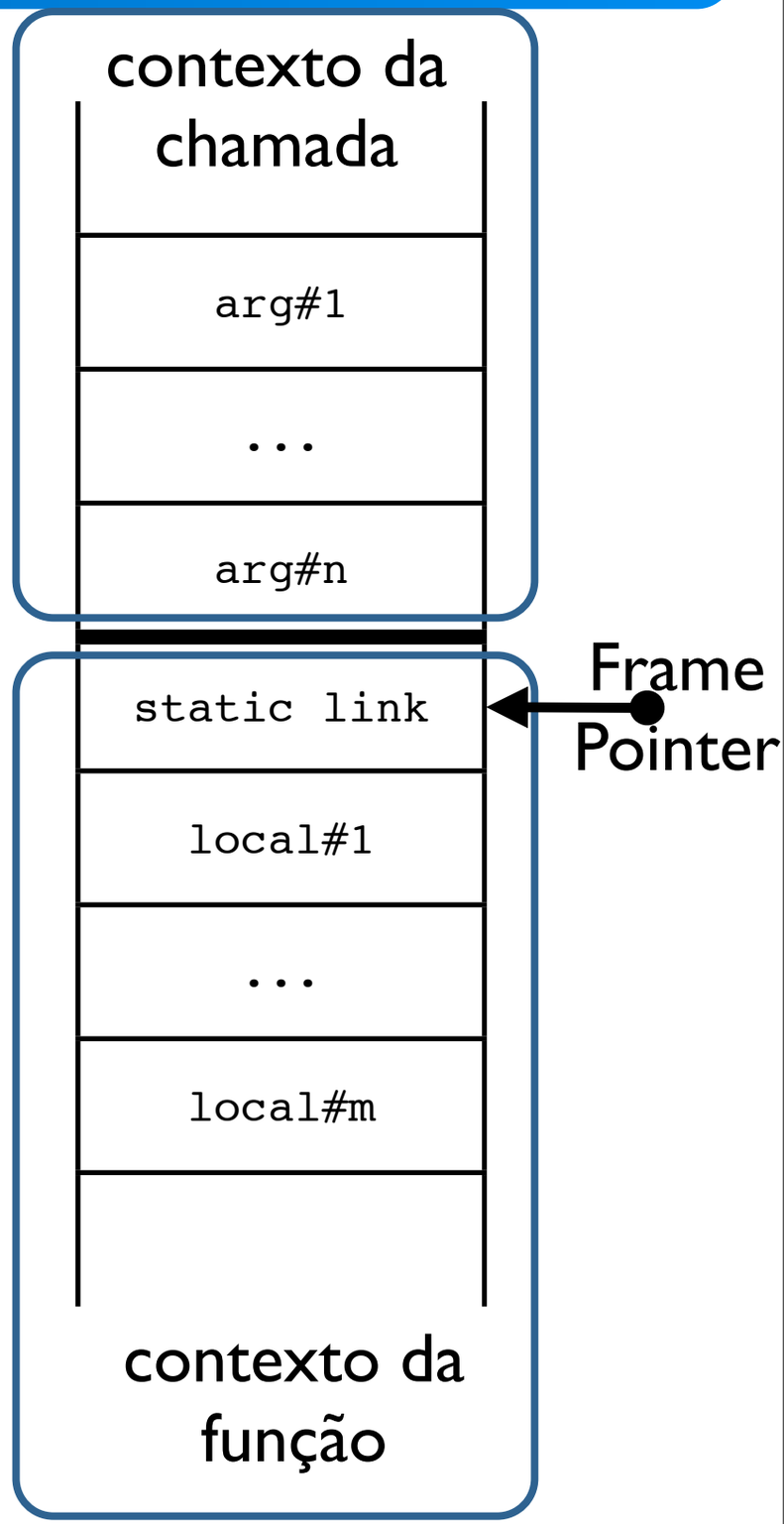


# Compilação de CALCF (Ideias Gerais)

- Cada abstracção é mapeada numa função CIL com um registo de activação próprio onde são guardados os valores correspondentes aos argumentos e às declarações locais.

- O registo de activação contém um link estático para o registo de activação da última ocorrência na pilha da abstracção envolvente estaticamente no programa. Posição 0 do registo de activação.

- Os argumentos são passados através do registo de activação, que é passado pelo mecanismo normal da linguagem CIL. Os valores são colocados no registo de activação **antes** da chamada. Numa pilha fazem parte do registo de activação anterior e têm deslocamentos negativos em relação ao FramePointer.



```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+y))
end
end
end
end
```

```
.method ... Main() {
.locals init (object[] table)

}

.method ... f0(...) {
.locals init (object[] table)

}

.method ... f1(...) {
.locals init (object[] table)

}

.method ... f2(...) {
.locals init (object[] table)

}

.method ... f3(...) {
.locals init (object[] table)

}

.method ... f4(...) {
.locals init (object[] table)

}
```

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+y))
end
end
end
end
```

```
.method .. Main() {
.locals init (object[] table)

}

.method .. f0(...) {
.locals init (object[] table)

}

.method .. f1(...) {
.locals init (object[] table)

}

.method .. f2(...) {
.locals init (object[] table)

}

.method .. f3(...) {
.locals init (object[] table)

}

.method .. f4(...) {
.locals init (object[] table)

}
```

# Compilação de CALCF

Os valores associados aos identificadores são guardados no âmbito local do método CIL correspondente. Ao separar o código em vários métodos não aninhados perde-se o acesso aos identificadores. As variáveis locais são apenas acessíveis dentro do método.

```
decl
  x = 1
  decl
    f = (fun y -> y+x)
  in
    decl
      g = (fun x -> f(x)+1)
      i = (fun y->(fun x -> x)(f(y)))
    in
      i(f(1))
    end
  end
end
```

```
.method ... static  ... void Main () {
  .locals (object[] table)
  ...
}
}
.method ... static ... object f0(object y)
{
  ldloc table // x ????
  ...
}
.method ... static ... object f1(object x)
{
  ldloc table // f ????
  ...
}
...
```

# Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end


  .locals (object[] table)
  ...
}
```

# Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end


  .locals (object[] table)
  ...
}
```

# Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end
```

```
  .locals (object[] table)
  ...
}
```

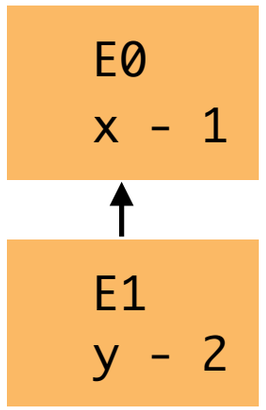
E0
x - 1

# Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end


  .locals (object[] table)
  ...
}
```

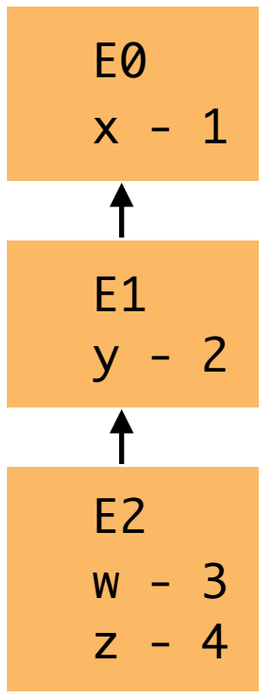


# Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end


  .locals (object[] table)
  ...
}
```

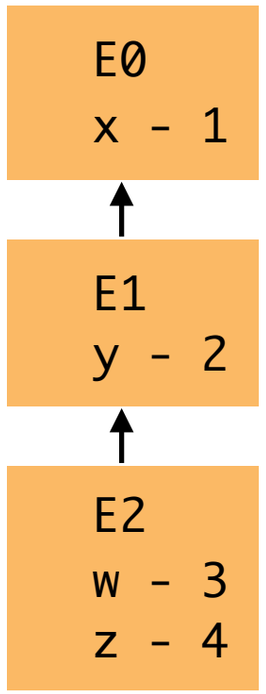


# Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end
```

```
  .locals (object[] table)
...
  ldloc table
  ldc.i4 3
  ldelem.ref // w
```

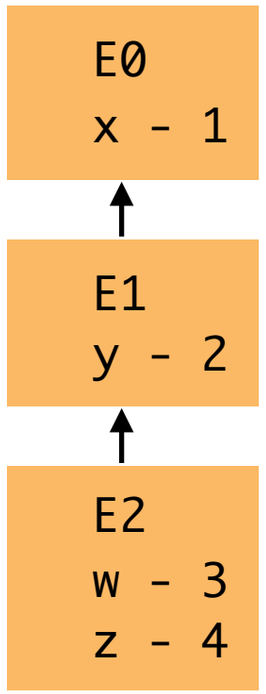


# Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end
```

```
  .locals (object[] table)
...
  ldloc table
  ldc.i4 4
  ldelem.ref // z
```

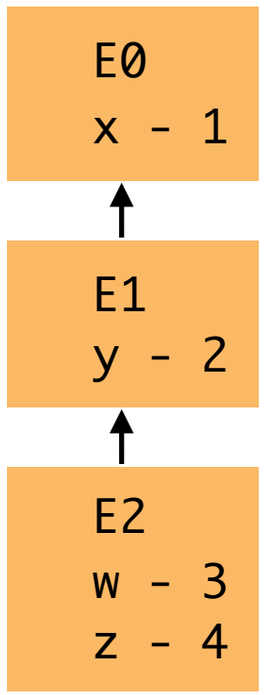


# Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end
```

```
.locals (object[] table)
   ...
ldloc table
ldc.i4 1
ldelem.ref // x
```

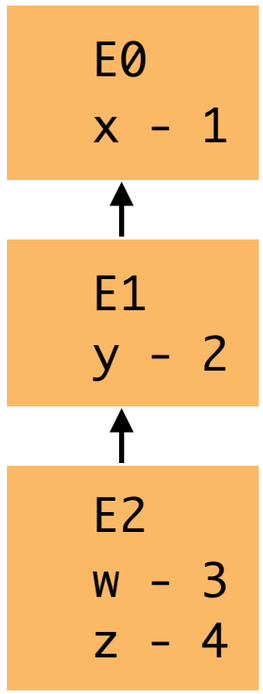


# Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end

  .locals (object[] table)
  ...
```

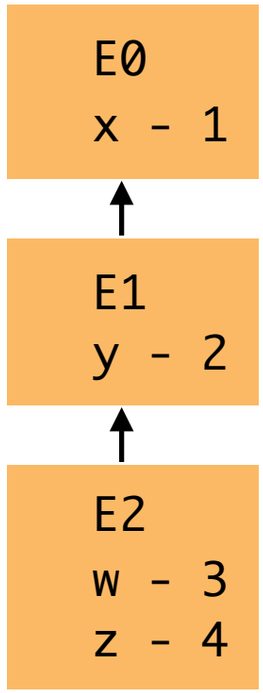


# Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end

  .locals (object[] table)
  ...
```

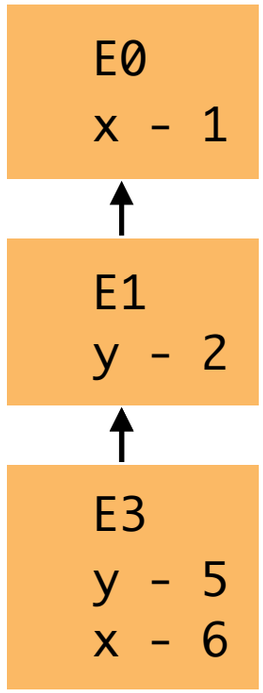


# Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end

  .locals (object[] table)
  ...
```



```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```

A primeira posição (0) fica reservada para o static link

Os endereços de identificadores começam em 1

```
.method ... Main() {
.locals init (object[] table)
}
```

| | x | f | g | h | i |
|---|---|---|---|---|---|

```
.method ... f0(...) {
.locals init (object[] table)
}
```

| | y | z |
|---|---|---|

```
.method ... f1(...) {
.locals init (object[] table)
}
```

| | w |
|---|---|

```
.method ... f2(...) {
.locals init (object[] table)
}
```

| | x | y |
|---|---|---|

```
.method ... f3(...) {
.locals init (object[] table)
}
```

| | y |
|---|---|

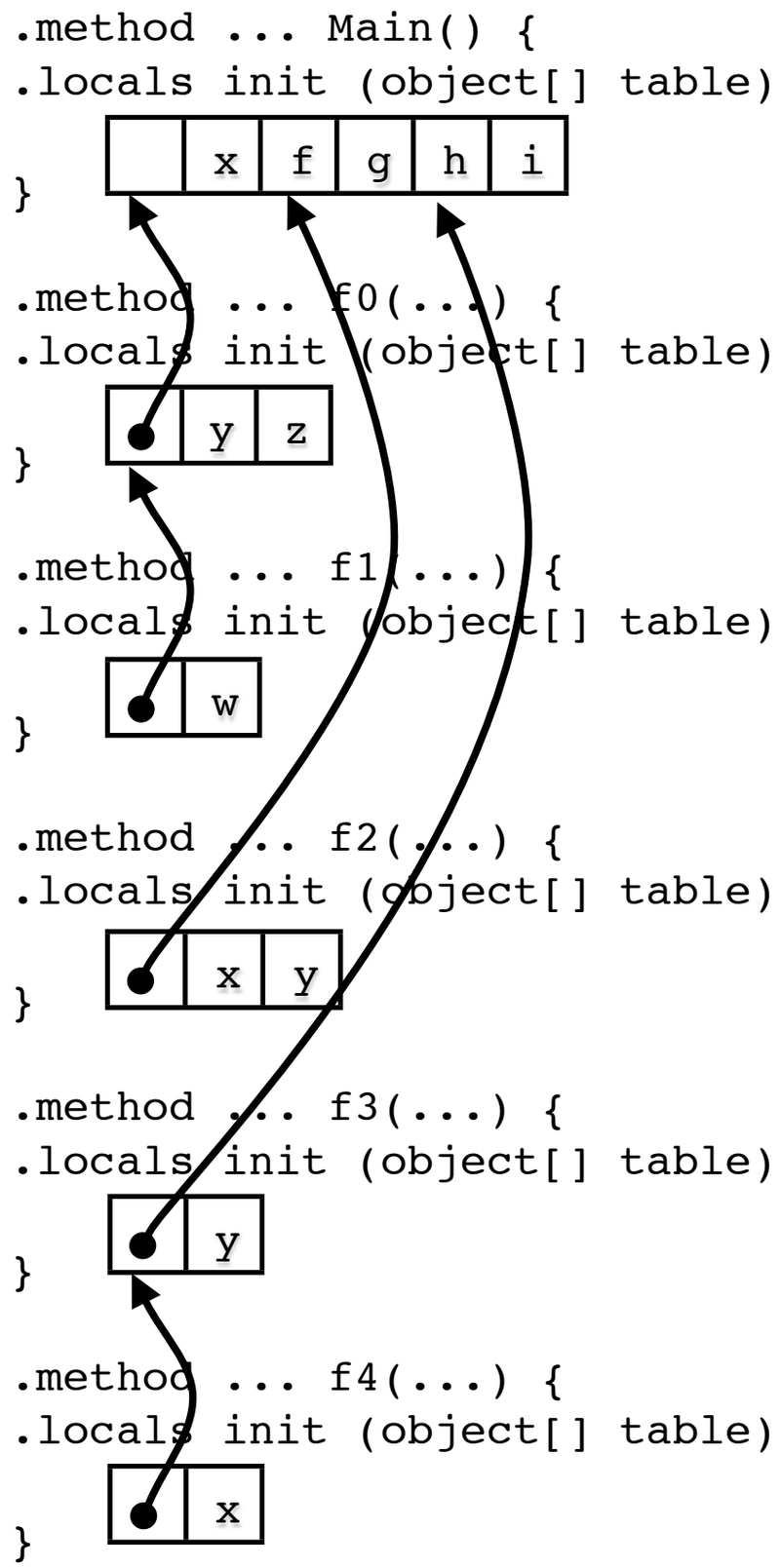
```
.method ... f4(...) {
.locals init (object[] table)
}
```

| | x |
|---|---|

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```

A primeira posição (0) fica reservada para o static link

Os endereços de identificadores começam em 1



```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```

- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```

```
E0    *
x - 1
```

- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
* começa um novo registo de activação
```

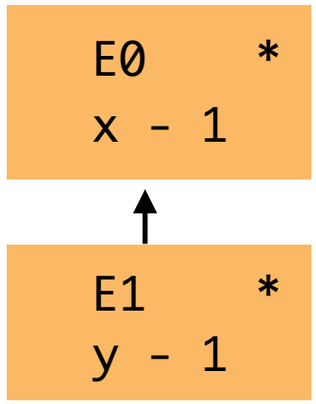
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```

```
E0    *
x - 1
```

- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
* começa um novo registo de activação
```

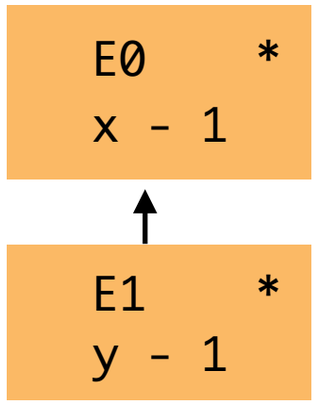
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

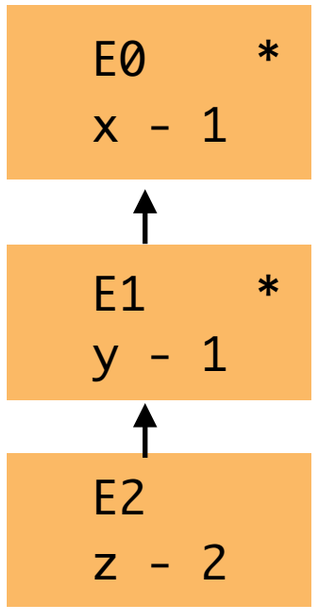
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

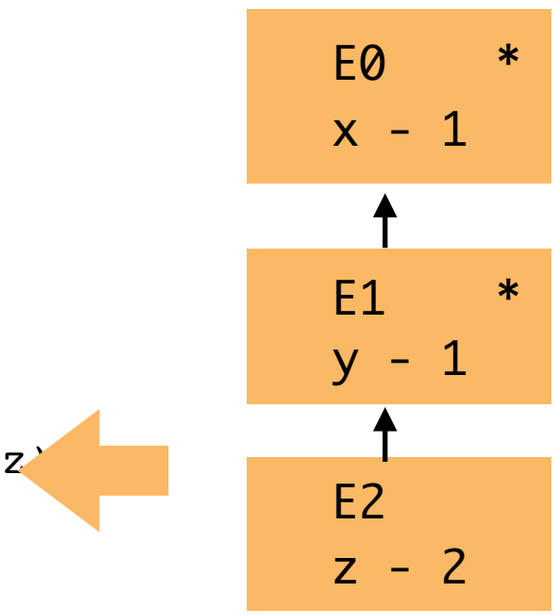
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

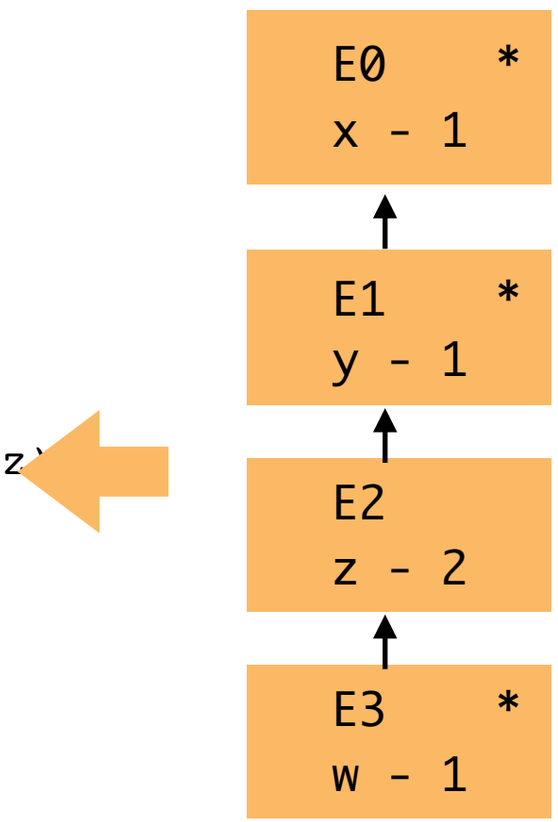
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

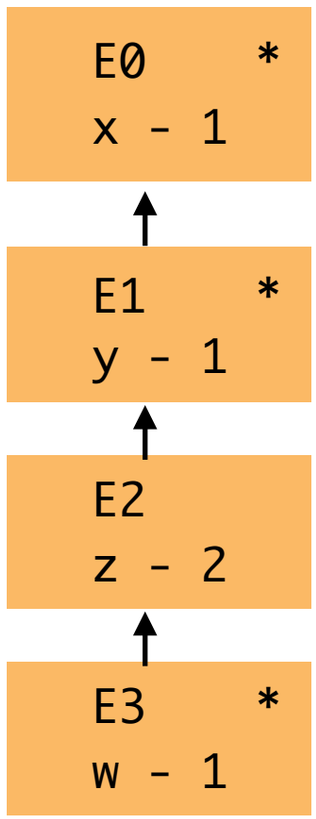
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

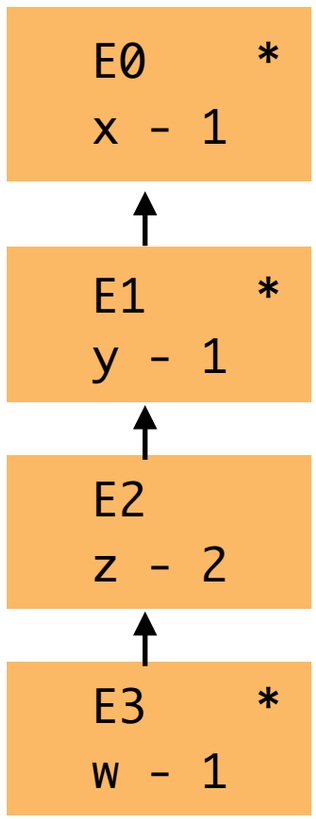
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```

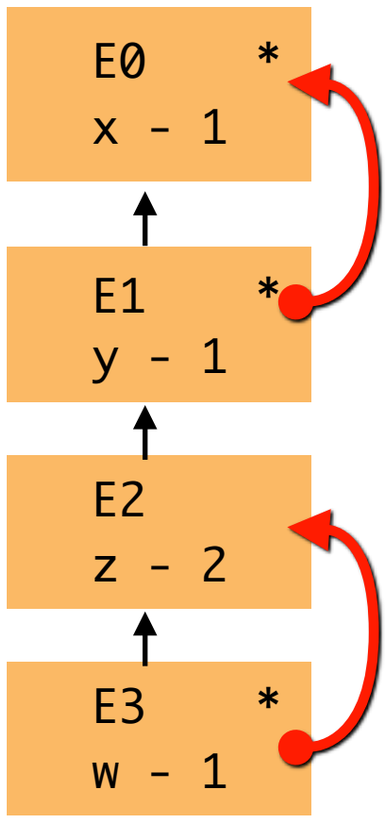



- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
w - (0,1)
x - (2,1)
y - (1,1)
z - (1,2)
```

* começa um novo registo de activação

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
w - (0,1)
x - (2,1)
y - (1,1)
z - (1,2)
```

* começa um novo registo de activação

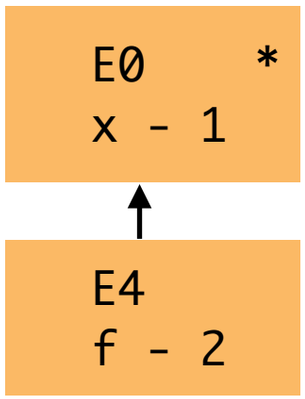
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```

```
E0    *
x - 1
```

- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
* começa um novo registo de activação
```

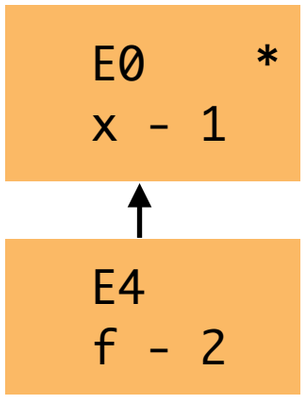
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

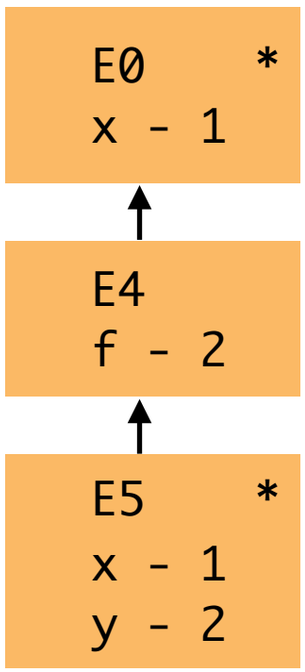
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

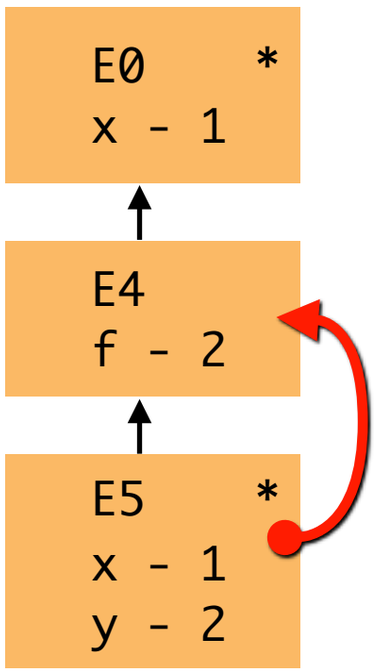
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```

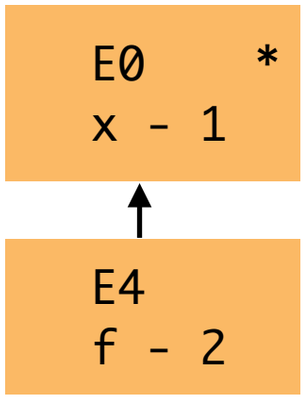



- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
f - (1,2)
x - (0,1)
y - (0,2)
```

* começa um novo registo de activação

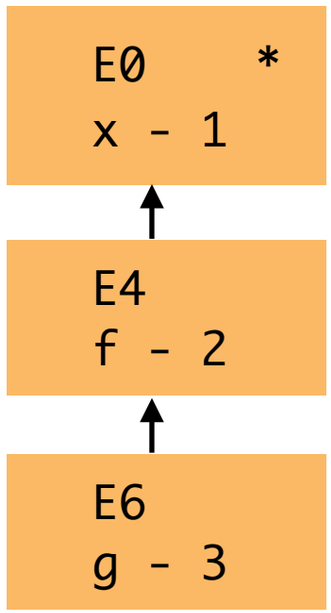
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```

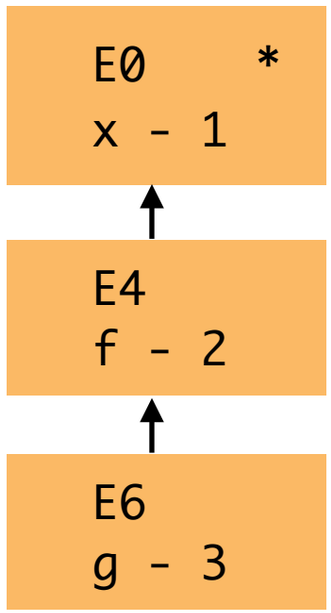



- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
f - (0,2)
```

* começa um novo registo de activação

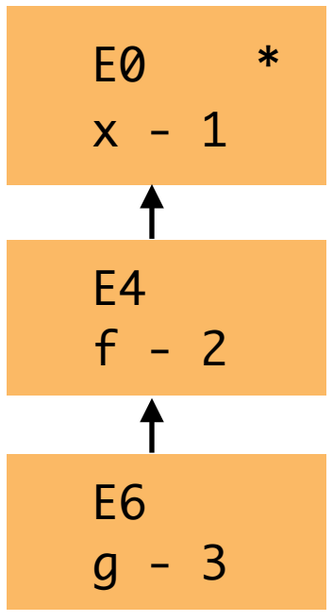
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

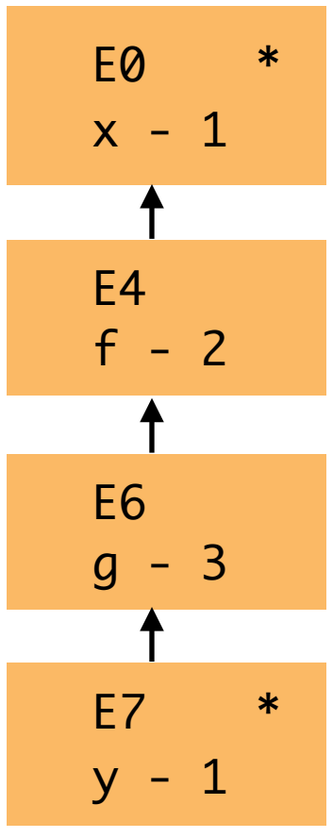
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

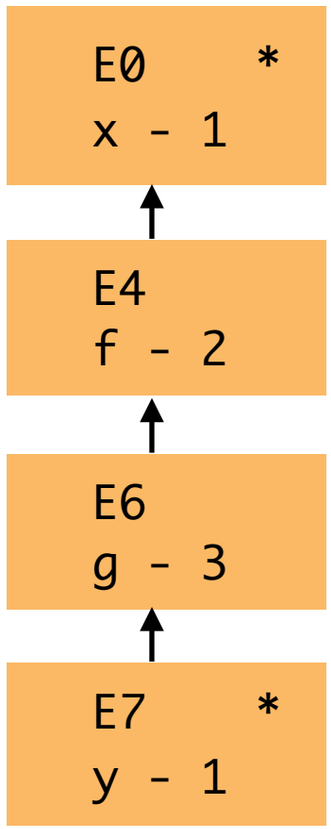
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

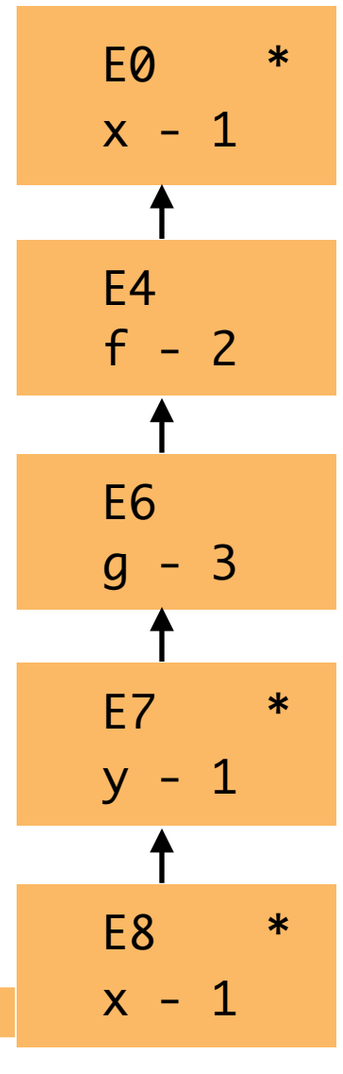
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

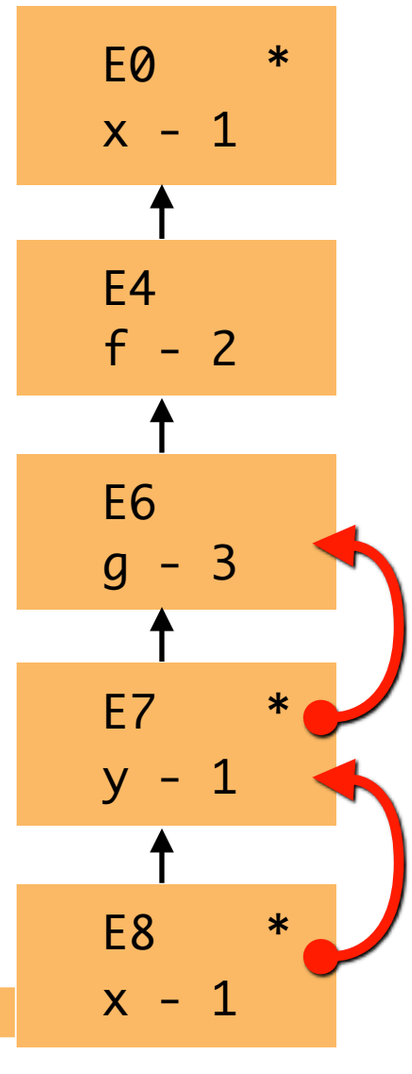
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```

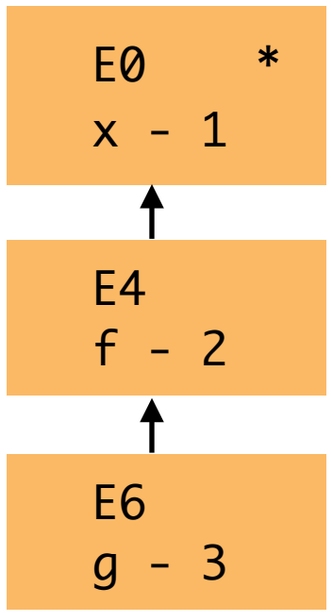



- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
g - (2,3)
x - (0,1)
y - (1,1)
```

* começa um novo registo de activação

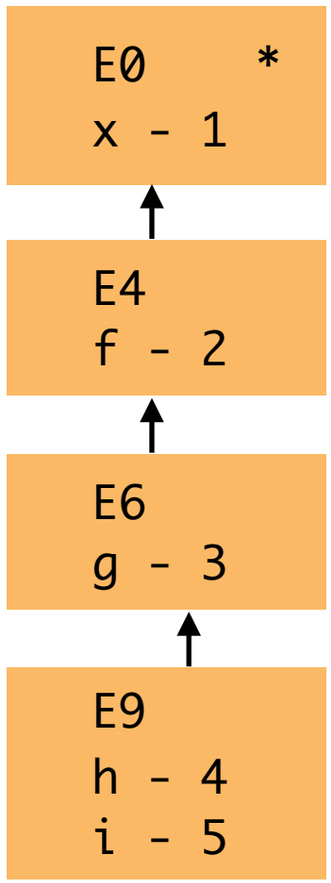
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

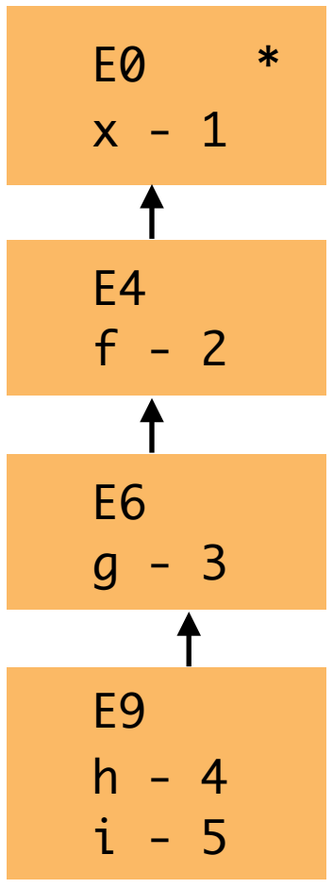
```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

* começa um novo registo de activação

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```




- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
i - (0,5)
h - (0,4)
x - (0,1)
```

* começa um novo registo de activação

# Ambiente "mutável"

- Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável:

```
Environ BeginScope()
```

  - Cria um novo nível **vazio**, onde serão colocadas as ligações de um novo âmbito local.
  - Não pode existir mais que uma ligação para um mesmo identificador no mesmo nível.

```
Environ EndScope()
```

  - Devolve o ambiente no estado anterior à última operação BeginScope().
  - Mas **não destrói** o último nível pois podem existir no contexto de execução fechos que o referem!

# Ambiente "mutável"

- Para suportar o endereçamento usando vários registos de activação
- Os ambientes devem distinguir dois tipos de situações

```
Environ BeginScope(boolean startsAR)
```

  - cria um novo nível local vazio, onde serão colocadas as novas ligações.
  - Regista se o novo nível corresponde ao início de um novo registo de activação.
  - Não pode existir mais que uma ligação para um mesmo identificador no mesmo nível.

```
int getNumDecls()
```

  - denota o número de declarações dentro do registo de activação corrente.

```
Pair<Integer,Integer> find(String id)
```

  - procura o identificador id no nível corrente e depois nos anteriores, o primeiro elemento do par denota o número de saltos entre registos de activação necessários, o segundo elemento denota o endereço guardado no ambiente.

# Ambiente "mutável"

**Atenção:** Os endereços dentro de um registo de activação não podem ser reutilizados (da mesma maneira que os registos de activação não podem ser desempilhados). Sugestão de implementação da classe ambiente:

```
class Env {
    ...
    Env up;
    boolean startFrame;
    int count;

    void assoc(String id, int addr) {
            ...
            incDecls();
    }

    private void incDecls() {
            if(startFrame) count++;
            else up.incDecls();
    }

    int getNumDecls() {
            if(startFrame) return count;
            else return up.getNumDecls();
    }
}
```

# Compilação de CALCF (Closures)

- As closures são valores de runtime que representam funções. Contêm uma referência para o código a executar e o acesso ao registo de activação da última instância da função que a envolve sintacticamente.

- Uma função pode produzir closures e por isso o seu registo de activação não pode ser destruído quando acaba a execução. Os registos de activação só são destruídos por "garbage collection".

- Nesse caso os registos de activação devem ser criados na heap e formam uma estrutura de dados chamada pilha esparguete (Spaghetti Stack).

- Implementação de closures em objectos no sistema de suporte à execução.

# Compilação de CALCF (Closures)

- Uma closure é composta por uma referência para o código de uma abstração e uma referência ao seu static link (um registo de activação).

```
class Closure {
    StackFrame stackFrame;
    IntPtr ftn;

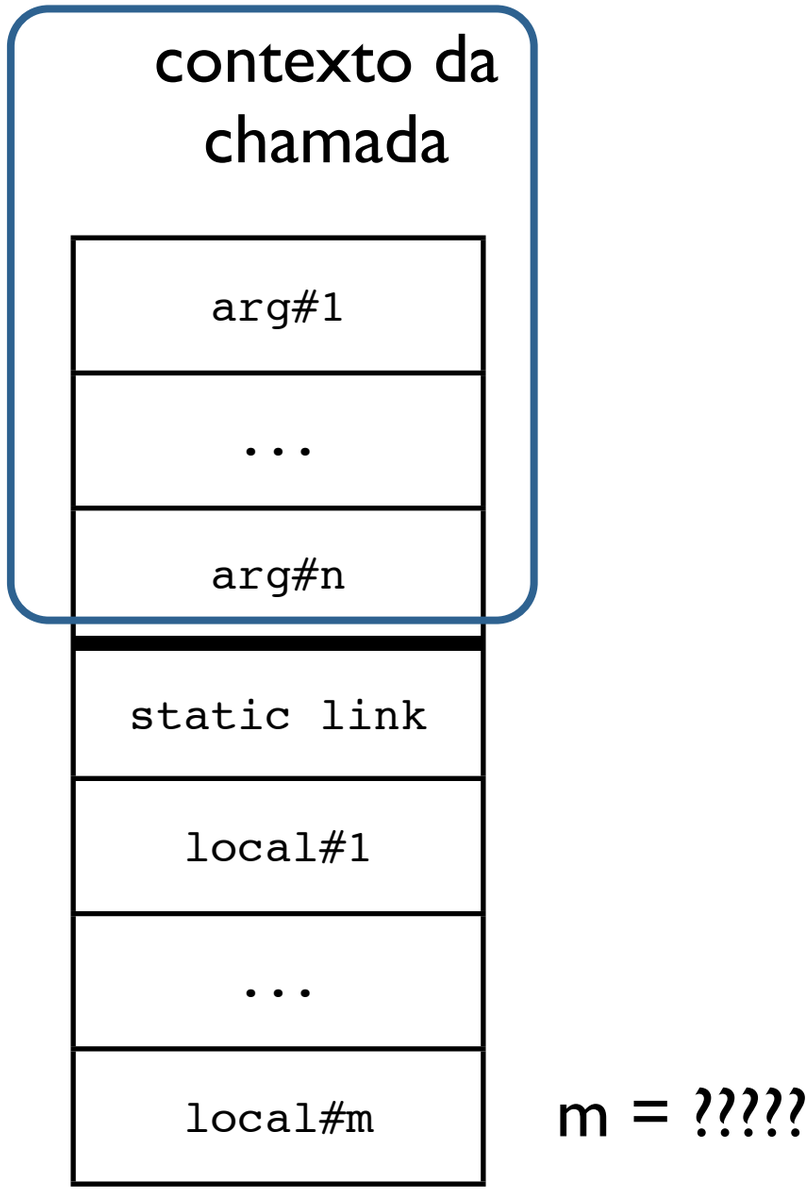
    Closure(StackFrame stackFrame, IntPtr ftn) {
        this.stackFrame = stackFrame;
        this.ftn = ftn;
    }
    StackFrame getSF() { return stackFrame; }
    IntPtr getFtn() { return ftn; }
}
```

- O registo de activação da closure é o registo de activação onde foi avaliada a definição da função ou procedimento. Neste ambiente de compilação é conhecido o nome da função CIL que lhe corresponde.

```
.locals init (object stackFrame)
...
ldloc stackFrame
ldftn object f0(object)
newobj instance void Closure::.ctor(class [StackFrame], native int)
```

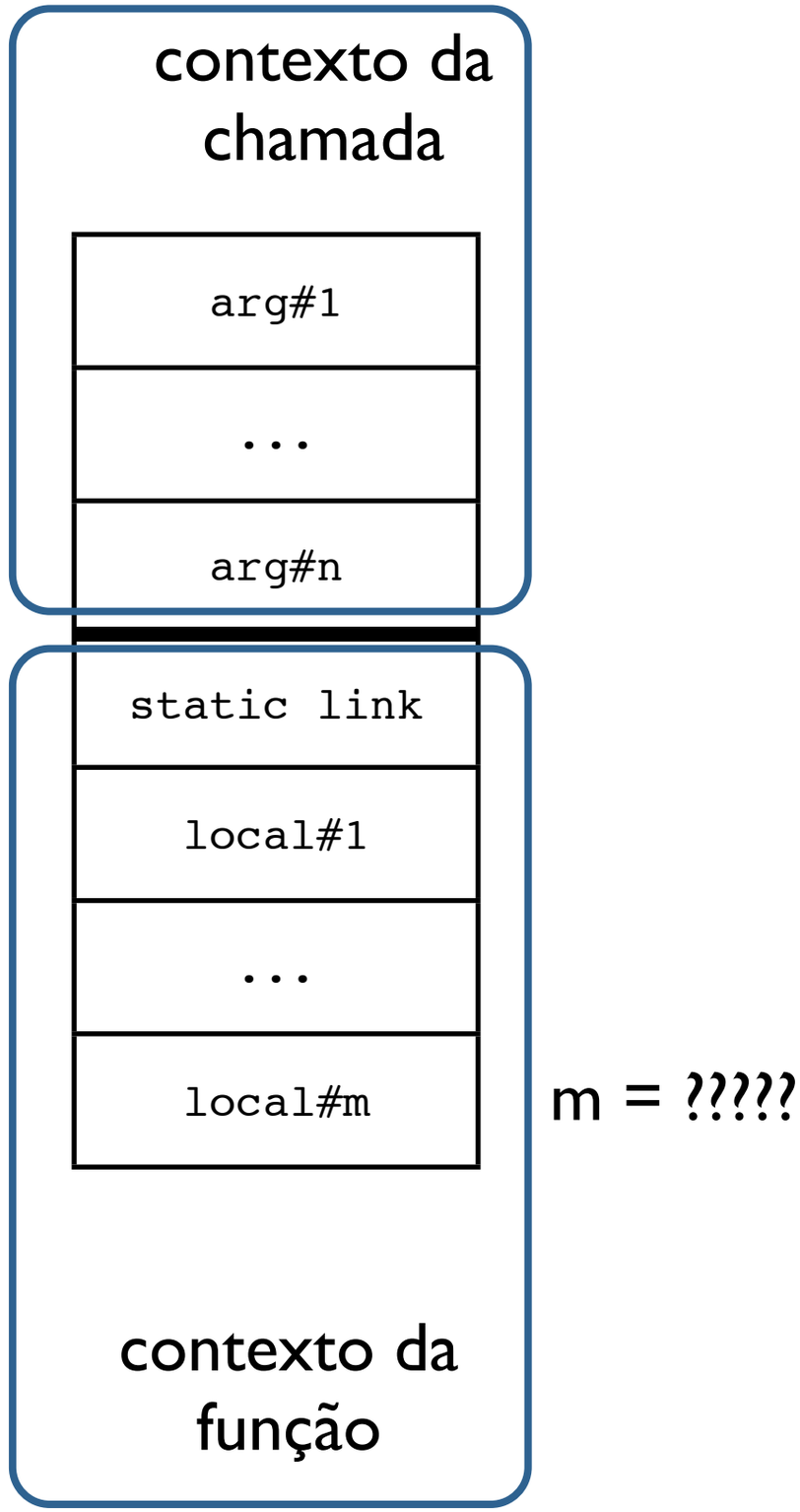
# Compilação de CALCF (passagem parâmetros)

- O registo de activação deve que ser criado e preenchido com os valores dos argumentos **antes** da chamada.
- No contexto da chamada sabemos calcular os argumentos e sabemos quantos são, mas...
- não sabemos quantas variáveis locais vão ser necessárias para a execução da função... (recorde-se que as funções são anónimas)
- logo, é necessária uma estrutura de dados mais dinâmica que um vector com um número fixo de posições.
- Suporte para a criação da memória para os argumentos e a memória para as variáveis locais em alturas diferentes (antes e depois da chamada da função).



# Compilação de CALCF (passagem parâmetros)

- O registo de activação deve que ser criado e preenchido com os valores dos argumentos **antes** da chamada.
- No contexto da chamada sabemos calcular os argumentos e sabemos quantos são, mas...
- não sabemos quantas variáveis locais vão ser necessárias para a execução da função...
  (recorde-se que as funções são anónimas)
- logo, é necessária uma estrutura de dados mais dinâmica que um vector com um número fixo de posições.
- Suporte para a criação da memória para os argumentos e a memória para as variáveis locais em alturas diferentes
  (antes e depois da chamada da função).



# Compilação de CALCF (passagem parâmetros)

- Suporte para a criação da memória para os argumentos e a memória para as variáveis locais em alturas diferentes (antes e depois da chamada da função).

  Sugestão de implementação:

```
class StackFrame {
   object staticlink; // indice 0

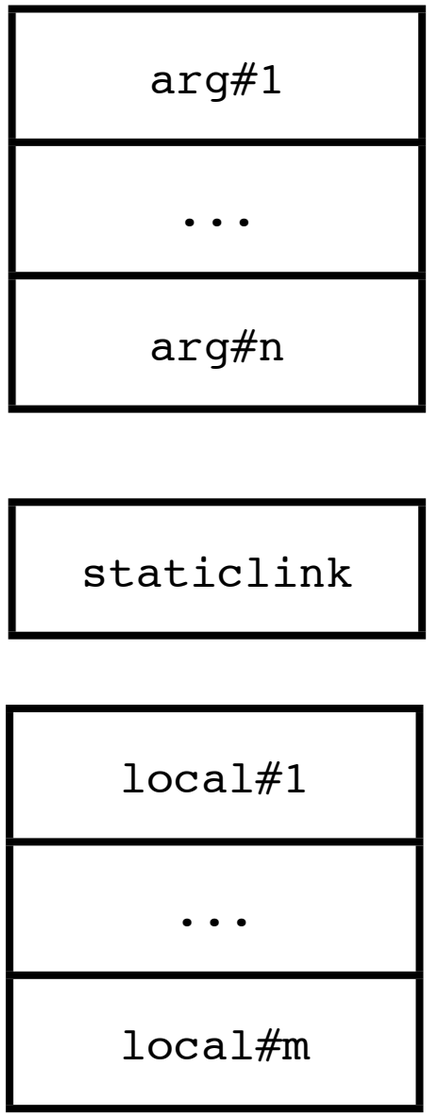
   object[] params;   // 1 a nParams
   int nParams;

   object[] locals;     // ... > nParams+1

   StackFrame(object SL) {...}

   void initArgs(int nParams) {...}
   void initLocals(init nLocals) {...}

   object get(int i) {...}
   void set(int i, object o) {...}
}
```



# Compilação de CALCF (passagem parâmetros)

- Suporte para a criação da memória para os argumentos e a memória para as variáveis locais em alturas diferentes (antes e depois da chamada da função).

Sugestão de implementação:

```
class StackFrame {
   object staticlink; // indice 0

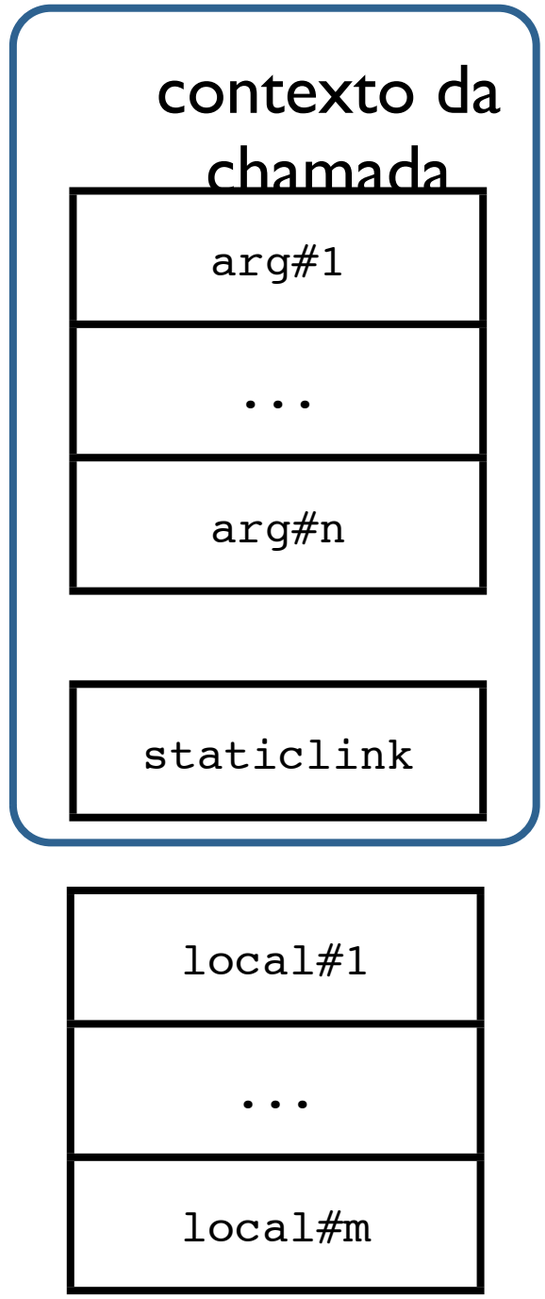
   object[] params;   // 1 a nParams
   int nParams;

   object[] locals;     // ... > nParams+1

   StackFrame(object SL) {...}

   void initArgs(int nParams) {...}
   void initLocals(init nLocals) {...}

   object get(int i) {...}
   void set(int i, object o) {...}
}
```



# Compilação de CALCF (passagem parâmetros)

- Suporte para a criação da memória para os argumentos e a memória para as variáveis locais em alturas diferentes (antes e depois da chamada da função).

Sugestão de implementação:

```
class StackFrame {
   object staticlink; // indice 0

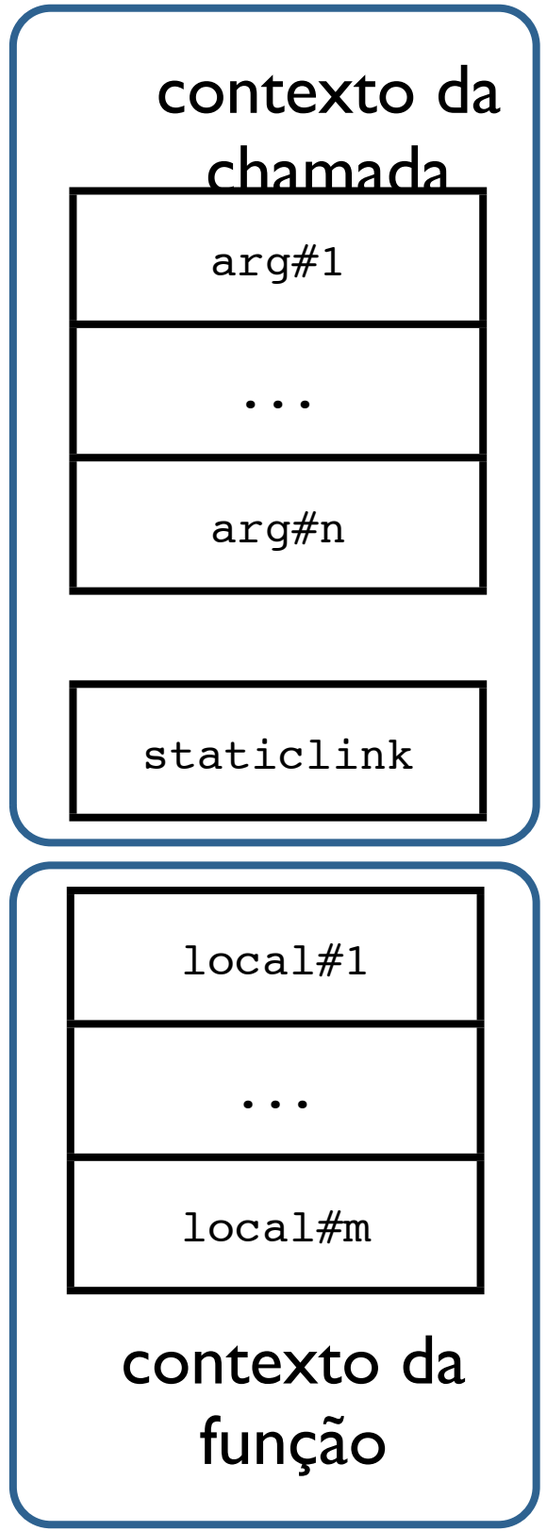
   object[] params;    // 1 a nParams
   int nParams;

   object[] locals;      // ... > nParams+1

   StackFrame(object SL) {...}

   void initArgs(int nParams) {...}
   void initLocals(init nLocals) {...}

   object get(int i) {...}
   void set(int i, object o) {...}
}
```



# Compilador de CALCF

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem CALCF:

$$\text{comp: } P \times ENV \rightarrow CodeSeq \times CodeSeq\ list$$

Dado um ambiente **env**

se *E* é da forma **num**( n ):

```
s = < ldc.i4 n > @ < box int32 >

comp(E,env) ≜ (s, [ ] )
```

se *E* é da forma **add**(E',E''):

```
(s1, f1) = comp(E',env);
(s2, f2) = comp(E'',env);
s ≜ s1 @ < unbox int32 > @ < ldobj int32 > @
    s2 @ < unbox int32 > @ < ldobj int32 > @
    <add> @ < box int32 >

comp(E,env) ≜ (s, f1 @ f2 )
```

# Compilador de CALCF

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem CALCF:

$$\text{comp}: P \times ENV \rightarrow CodeSeq \times CodeSeq\ list$$

Dado um ambiente **env**

se E é da forma **id**( s ): (jumps,offset) = *env*.find(s);

```
s = <ldloc stackframe>
@<ldc.i4 0>
@<callvirt instance object StackFrame::get(int32)>
... // n jumps
@<ldc.i4 0>
@<callvirt instance object StackFrame::get(int32)>
@<ldc.i4 offset>
@<callvirt instance object StackFrame::get(int32)>
```

**comp**(E, *env*) ≜ ( s, [ ] )

# Compilador de CALCF

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem CALCF:

$$comp: P \times ENV \rightarrow CodeSeq \times CodeSeq\ list$$

Dado um ambiente **env**

se E é da forma **fun**(id,E):  n = fresh_fun_identifier()

*env* = *env*.BeginScope(**true**) // true => starts a new Stackframe...

*env*.Assoc(id, env.getNumDecls());

( sE, f ) = **comp**(E,*env*);

*env* = *env*.EndScope();

```
s =    <ldftn object fn(object)> @ <ldloc stackframe> @
       <newobj instance void Closure::.ctor(object,object)>
```

**comp**(E,*env*) ≜ ( s , [(“fn”,sE)] @ f )

# Compilador de CALCF

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem CALCF:

$$\text{comp}: P \times ENV \rightarrow CodeSeq \times CodeSeq\ list$$

Dado um ambiente **env**

se E é da forma **call**(E1,E2):

```
( s1, f1 ) = comp(E1, env);
( s2, f2 ) = comp(E1, env);
s = s1 @ ...
```

1. Construir um StackFrame com o static link da closure
2. Colocar os argumentos no topo da pilha e depois no StackFrame
3. Retirar a referência para a função da Closure que se encontra na pilha
4. Chamar a função passando o StackFrame já preenchido

**comp**(E,env) ≜ `(s, f1@f2 )`

# Compilador de CALCF

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem CALCF:

$$\text{comp}: P \times ENV \rightarrow CodeSeq \times CodeSeq\ list$$

Dado um ambiente **env**

se E é da forma **call**(E1,E2):

```
( s1, f1 ) = comp(E1, env);
( s2, f2 ) = comp(E1, env);
s = s1 @
```

1. Construir um StackFrame com o static link da closure

```
<dup> @
<callvirt instance Closure::getSF()> @
<newobj instance void StackFrame::.ctor(object)> @ ...
```

2. Colocar os argumentos no topo da pilha e depois no StackFrame
3. Retirar a referência para a função da Closure que se encontra na pilha
4. Chamar a função passando o StackFrame já preenchido

**comp**(E,env) ≜ `(s, f1@f2 )`

# Compilador de CALCF

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem CALCF:

comp: Exp x ENV → CodeSeq x CodeSeq list

Dado um ambiente **env**

se E é da forma **call**(E1,E2):

```
( s1, f1 ) = comp(E1, env);
( s2, f2  ) = comp(E1, env);
s = s1 @
```

1. Construir um StackFrame com o static link da closure

```
<dup> @
<callvirt instance Closure::getSF()> @
<newobj instance void StackFrame::.ctor(object)> @
```

2. Colocar os argumentos no topo da pilha e depois no StackFrame

```
<dup> @ <ldc.i4 1> @ <callvirt void initArgs(int32)> @
<dup> @ <ldc.i4 1> @ s2 @
<callvirt void StackFrame::set(int32,object)> @ ...
```

3. Retirar a referência para a função da Closure que se encontra na pilha
4. Chamar a função passando o StackFrame já preenchido

**comp**(E,env) ≜ `(s, f1@f2 )`

# Compilador de CALCF

Dado um ambiente **env**

se E é da forma **call**(E1,E2):

```
( s1, f1 ) = comp(E1, env);
( s2, f2  ) = comp(E1, env);
s = s1 @
```

1. Construir um StackFrame com o static link da closure

```
<dup> @
<callvirt instance Closure::getSF()> @
<newobj instance void StackFrame::.ctor(object)> @
```

2. Colocar os argumentos no topo da pilha e depois no StackFrame

```
<dup> @ <ldc.i4 1> @ <callvirt void initArgs(int32)> @
<dup> @ <ldc.i4 1> @ s2 @
<callvirt void StackFrame::set(int32,object)> @
```

3. Retirar a referência para a função da Closure que se encontra na pilha

3.1 trocar os 2 valores que estão no topo da pilha para que a closure fique acessível

3.2 obter o apontador para a função a partir da Closure.

```
<callvirt instance object Closure::getFtn()> @ ...
```

4. Chamar a função passando o StackFrame já preenchido

**comp**(E,env) ≜ `(s, f1@f2 )`

# Compilador de CALCF

Dado um ambiente **env**

se E é da forma **call**(E1,E2):

```
( s1, f1 ) = comp(E1, env);
( s2, f2  ) = comp(E1, env);
s = s1 @
```

1. Construir um StackFrame com o static link da closure

```
<dup> @
<callvirt instance Closure::getSF()> @
<newobj instance void StackFrame::.ctor(object)> @
```

2. Colocar os argumentos no topo da pilha e depois no StackFrame

```
<dup> @ <ldc.i4 1> @ <callvirt void initArgs(int32)> @
<dup> @ <ldc.i4 1> @ s2 @
<callvirt void StackFrame::set(int32,object)> @
```

3. Retirar a referência para a função da Closure que se encontra na pilha

3.1 trocar os 2 valores que estão no topo da pilha para que a closure fique acessível

```
<stloc tmp> @
<stloc tmp2> @
<ldloc tmp> @
<ldloc tmp2> @
```

3.2 obter o apontador para a função a partir da Closure.

```
<callvirt instance object Closure::getFtn()> @ ...
```

4. Chamar a função passando o StackFrame já preenchido

**comp**(E,env) ≜ `(s, f1@f2 )`

Dado um ambiente **env**

se E é da forma **call**(E1,E2):

```
( s1, f1 ) = comp(E1, env);
( s2, f2  ) = comp(E1, env);
s = s1 @
```

1. Construir um StackFrame com o static link da closure

```
<dup> @
<callvirt instance Closure::getSF()> @
<newobj instance void StackFrame::.ctor(object)> @
```

2. Colocar os argumentos no topo da pilha e depois no StackFrame

```
<dup> @ <ldc.i4 1> @ <callvirt void initArgs(int32)> @
<dup> @ <ldc.i4 1> @ s2 @
<callvirt void StackFrame::set(int32,object)> @
```

3. Retirar a referência para a função da Closure que se encontra na pilha

3.1 trocar os 2 valores que estão no topo da pilha para que a closure fique acessível

```
<stloc tmp> @
<stloc tmp2> @
<ldloc tmp> @
<ldloc tmp2> @
```

3.2 obter o apontador para a função a partir da Closure.

```
<callvirt instance object Closure::getFtn()> @
```

4. Chamar a função passando o StackFrame já preenchido

```
<calli object(object)>
```

**comp**(E,env) ≜ (s, f1@f2 )

# Compilação de CALCF (exemplo)

## preâmbulo e estrutura da assembly

```
(fun x -> decl y = 2 in x+y)(2)
```

```
.assembly Calc {}
.assembly extern Runtime {}
.module Calc.exe
.method public static  hidebysig default void Main ()  cil managed
{
  .entrypoint
  .locals init (object stackframe, object tmp, object tmp2)
  ldnull
  newobj instance void [Runtime]StackFrame::.ctor(class [Runtime]StackFrame)
  stloc stackframe
  ... // closure, chamada & unbox
  call void class [mscorlib]System.Console::Write(int32)
  ret
}
.method public static  hidebysig default object f1(object) cil managed
{
  .locals init (object stackframe, object tmp, object tmp2)
  ldarg 0
  stloc stackframe
   ldloc stackframe
   ldc.i4 1
   callvirt instance void [Runtime]StackFrame::initLocals(int32)
  ... // corpo da função
  ret
}
```

# Compilação de CALCF (exemplo)

## definição da abstração

```
(fun x -> decl y = 2 in x+y)(2)
```

```
...
.method public static  hidebysig default void Main ()  cil managed
{
  .entrypoint
  .locals init (object stackframe, object tmp, object tmp2)
  ... // preambulo
  ldloc stackframe
  ldftn object f1(object)
  newobj instance void [Runtime]Closure::.ctor(class [Runtime]StackFrame,object)
  ... // chamada, unbox & print
}
.method public static  hidebysig default object f1(object) cil managed
{
  .locals init (object stackframe, object tmp, object tmp2)
  ... // preambulo, initLocals
  ldloc stackframe // decl y = ...
  ldc.i4 2
  ldc.i4 2
  box int32
  callvirt instance void [Runtime]StackFrame::set(int32,object)

  ldloc stackframe // x
  ldc.i4 1
  callvirt instance object [Runtime]StackFrame::get(int32)
  ... // unbox, y, unbox, add & box
  ret
}
```

# Compilação de CALCF (exemplo)

## chamada da função...

```
(fun x -> decl y = 2 in x+y)(2)
```

```
...
.method public static  hidebysig default void Main ()  cil managed
{
  .entrypoint
  .locals init (object stackframe, object tmp, object tmp2)
  ... // Closure no topo da pilha...
  dup
  callvirt instance class [Runtime]StackFrame [Runtime]Closure::getSF()
  newobj instance void [Runtime]StackFrame::.ctor(class [Runtime]StackFrame)
  dup
  ldc.i4 1
  callvirt instance void [Runtime]StackFrame::initArgs(int32)
  dup
  ldc.i4 1
  ldc.i4 2
  box int32
  callvirt instance void [Runtime]StackFrame::set(int32,object)
  stloc tmp
  stloc tmp2
  ldloc tmp
  ldloc tmp2
  callvirt instance object [Runtime]Closure::getFtn()
  calli object(object)
  ... // unbox & print
```

# Definições Recursivas

- Na construção de declaração local de identificadores

```
decl id = E1 in E2 end
```

**o nome `id` não é ligado às ocorrências do mesmo nome em $E_1$**

- Por exemplo, na expressão

```
decl x=1 in
  decl x=x+1 in
    x+1
  end
end
```

a ocorrência de x na inicialização do segundo `decl` está ligada à primeira declaração `x = 1`.

- A segunda definição de x não é "recursiva" !!

# Definições Recursivas

- Uma definição recursiva é uma declaração onde o identificador declarado pode ocorrer dentro do corpo da definição:

```
declrec Sum = (fun x → if x=0 then 1
                       else x + Sum(x-1))
in
  Sum(10)
end
```

Mais rigorosamente: numa declaração recursiva a ocorrência ligante de um identificador também liga as ocorrências livres do mesmo identificador na expressão de inicialização.

- **Problema: como introduzir e como definir a semântica de definições recursivas (de funções) ?**

# A Linguagem RECF

- A linguagem RECF é a extensão da linguagem CALCF com expressões condicionais e definições recursivas.

```
num:      integer → RECF
var:      string → RECF
add:      RECF × RECF → RECF
...
if:       RECF × RECF × RECF → RECF
declrec:  string × RECF × RECF → RECF
fun:      string × RECF → RECF
call:     RECF × RECF → RECF
```

# A Linguagem RECF

- Exemplo de programa em RECF:

```
declrec
   fact = fun n → if n then n*fact(n-1) else 1 end
in
   fact(2)
end
```

- Abreviaturas:

$\texttt{decl id} = E_1 \texttt{ in } E_2 \triangleq \texttt{call((fun id} \rightarrow E_2), E_1)$

$E_1(E_2) \triangleq \texttt{call}(E_1, E_2)$

$\texttt{if } E_1 \texttt{ then } E_1 \texttt{ else } E_3 \triangleq \texttt{if}(E_1, E_2, E_3)$

# A Linguagem RECF

- Exemplo de programa em RECF:

```
declrec
   fact = fun n → if n then n*fact(n-1) else 1 end
in
   fact(2)
end
```

- Qual o ambiente no qual a abstracção deve ser avaliada/definida?
  Note que:
  - fact é uma ocorrência livre na abstracção
  - O identificador fact deverá estar definido no ambiente activo no momento em que a abstracção for avaliada
  - Por outro lado, o valor a associar a fact nesse mesmo ambiente deverá ser a própria função.

# A Linguagem RECF

- Exemplo de programa em RECF:

```
declrec
   fact = fun n → if n then n*fact(n-1) else 1 end
in
   fact(2)
end
```

- Qual o ambiente no qual a abstracção deve ser avaliada/definida?

  - As definições recursivas introduzem uma "circularidade" na construção do ambiente:

  - O ambiente E resultante da declaração de fact deverá ter uma ligação `fact → closure(n, if ... end , E)` que associe ao nome `fact` um fecho cuja componente ambiente é o próprio `E`

  - Em geral, um ambiente E deverá pode conter ligações entre identificadores e fechos com referências para (partes de) o próprio ambiente E

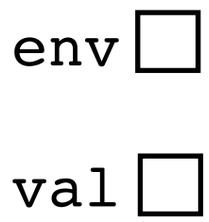
# Ambiente Mutável (revisitado)

- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.



# Ambiente Mutável (revisitado)

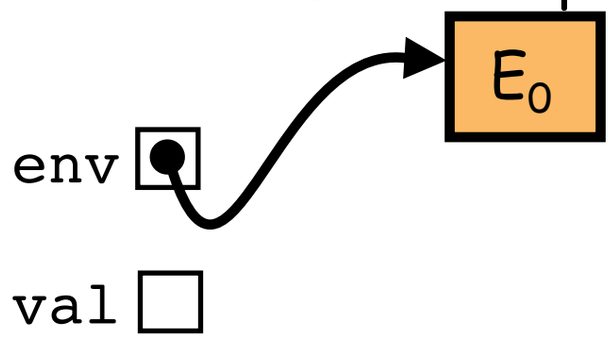
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
```



# Ambiente Mutável (revisitado)

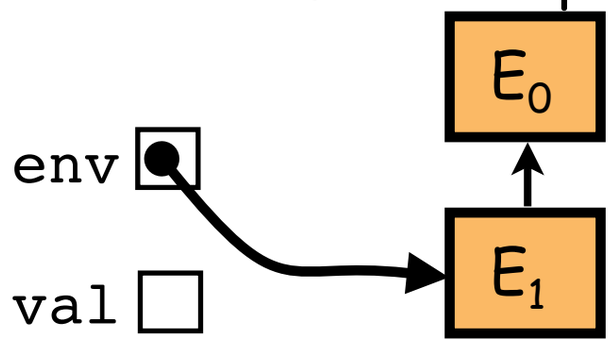
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = BeginScope();
```



# Ambiente Mutável (revisitado)

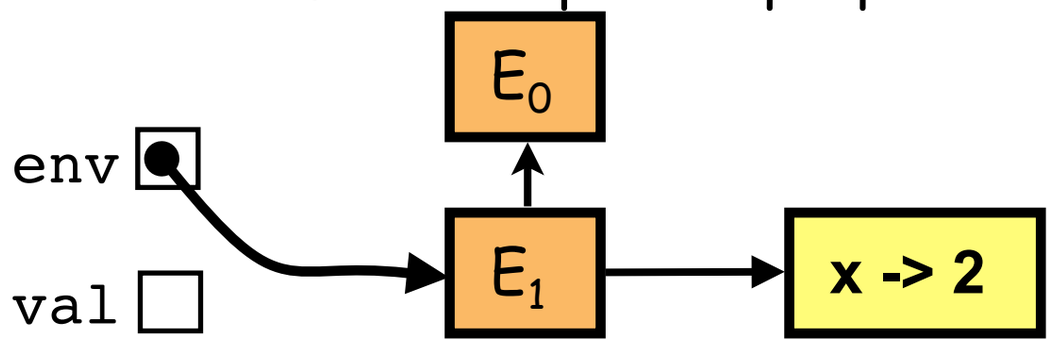
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = BeginScope();
env.Assoc("x", 2);
```



# Ambiente Mutável (revisitado)

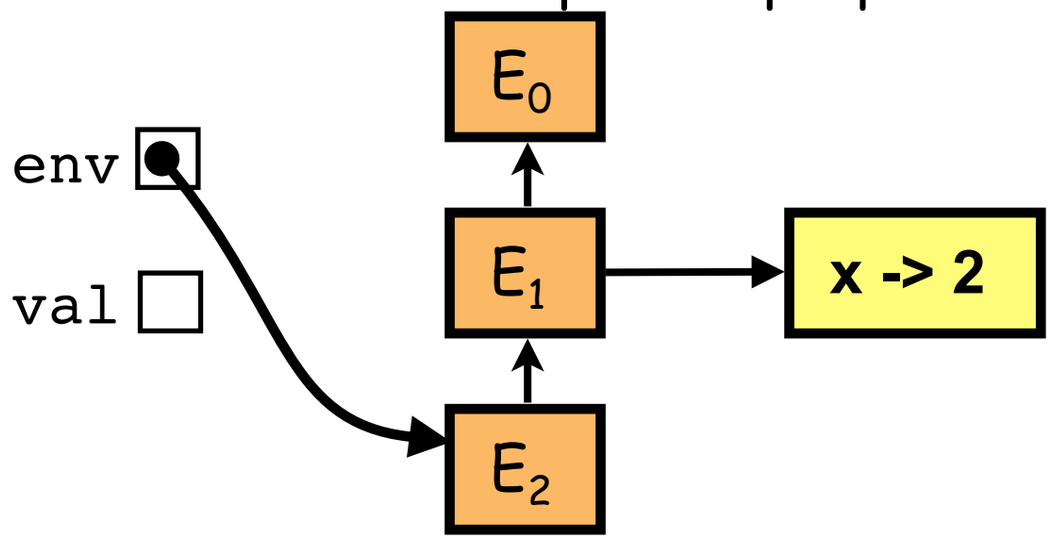
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = BeginScope();
env.Assoc("x", 2);
env = env.BeginScope();
```



# Ambiente Mutável (revisitado)

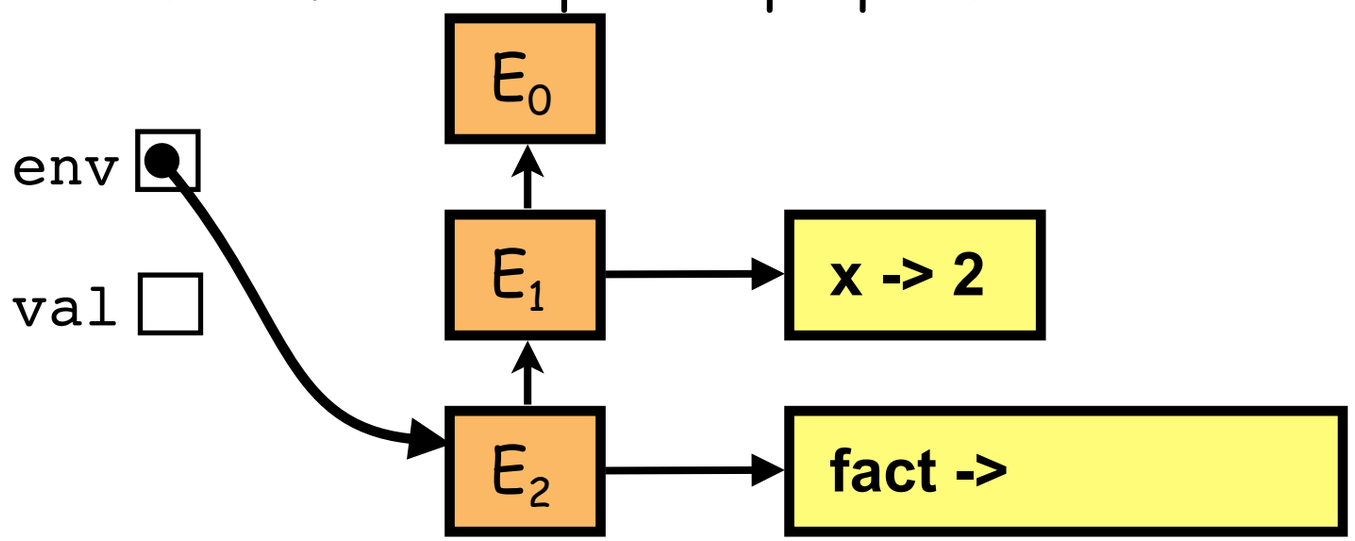
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = BeginScope();
env.Assoc("x", 2);
env = env.BeginScope();
env.Assoc("fact", null);
```



# Ambiente Mutável (revisitado)

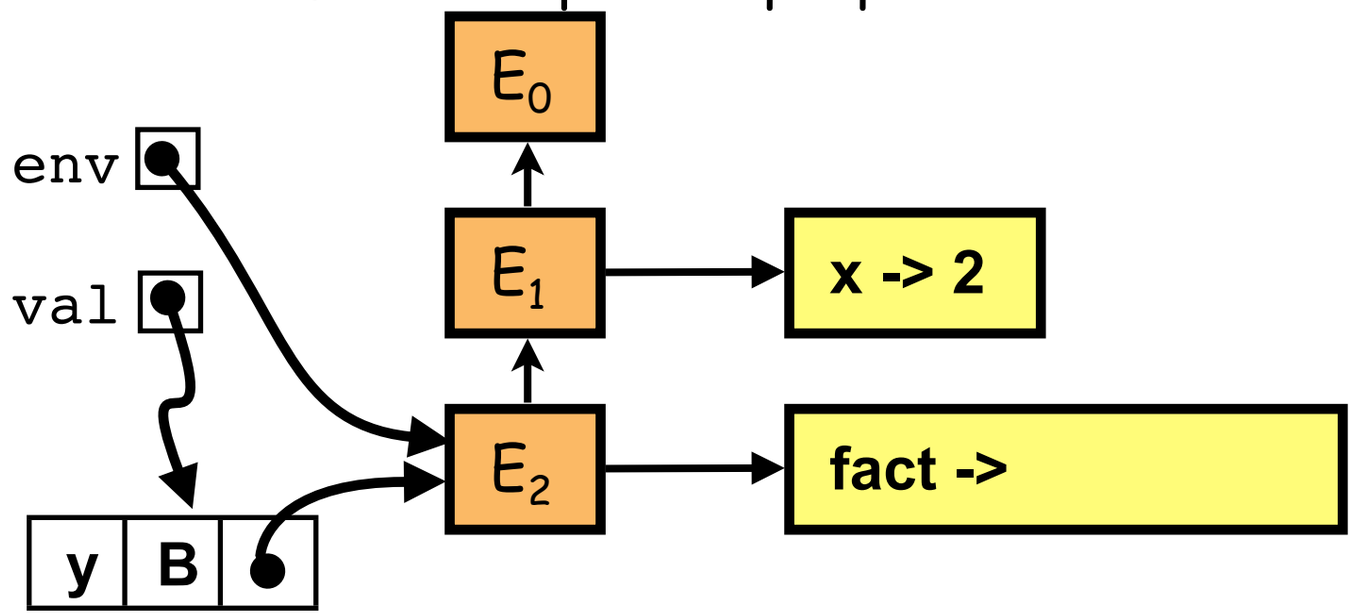
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = BeginScope();
env.Assoc("x", 2);
env = env.BeginScope();
env.Assoc("fact", null);
val = closure("y", B, env);
```



# Ambiente Mutável (revisitado)

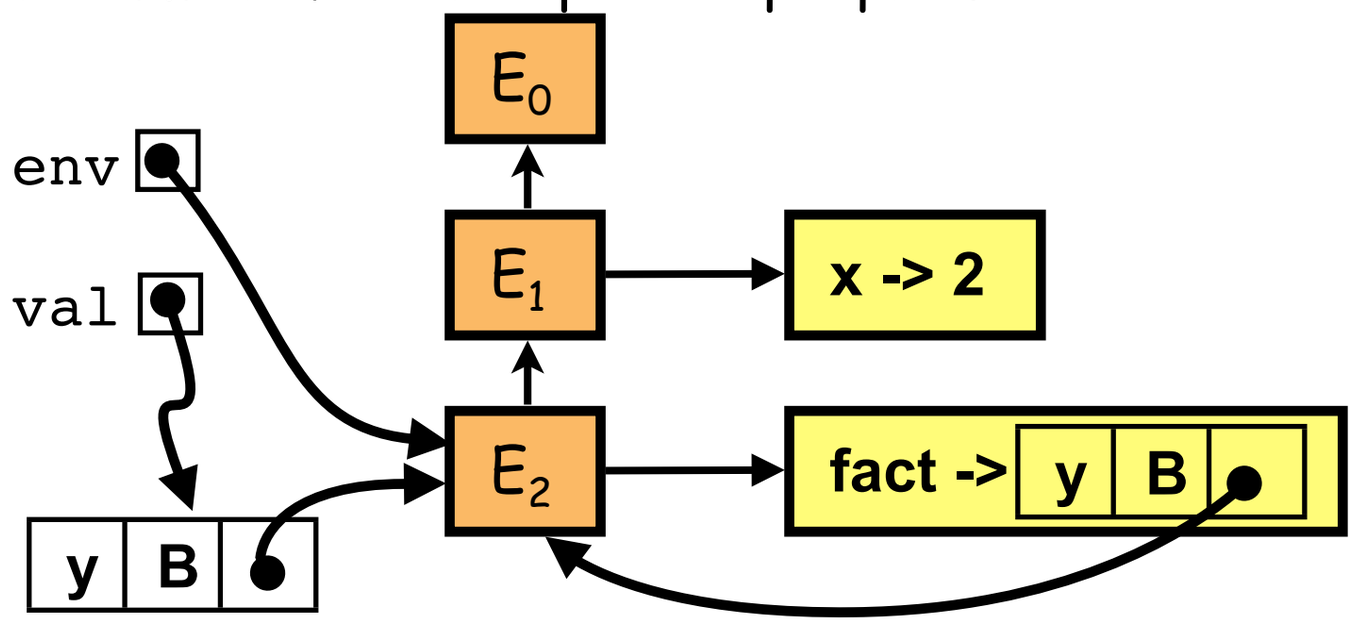
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = BeginScope();
env.Assoc("x", 2);
env = env.BeginScope();
env.Assoc("fact", null);
val = closure("y", B, env);
env.Update("fact", val);
```



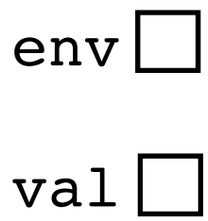
# Ambiente Mutável (revisitado)

- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.



# Ambiente Mutável (revisitado)

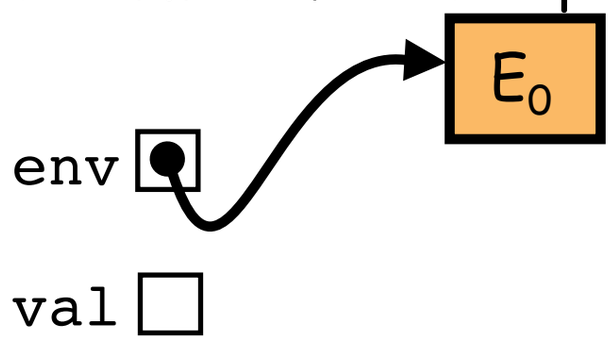
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
```



# Ambiente Mutável (revisitado)

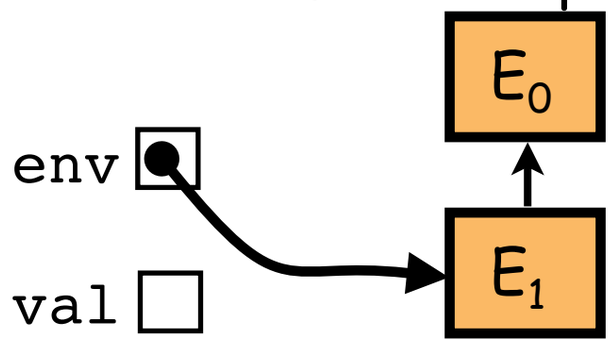
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = env.BeginScope();
```



# Ambiente Mutável (revisitado)

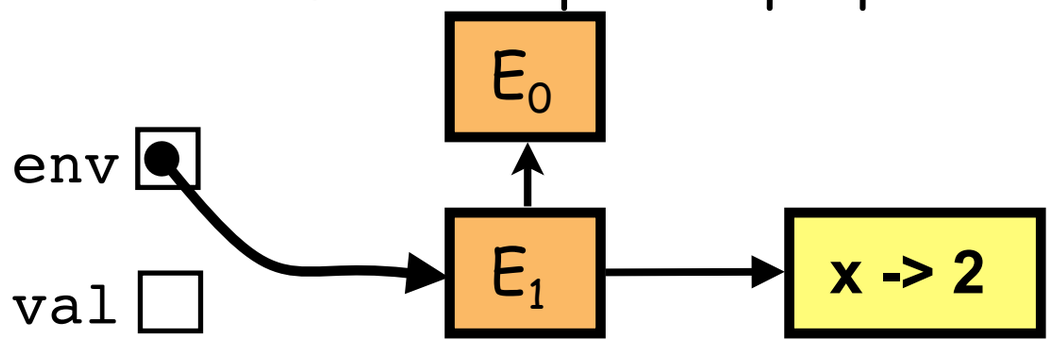
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = env.BeginScope();
env.Assoc("x", 2);
```



# Ambiente Mutável (revisitado)

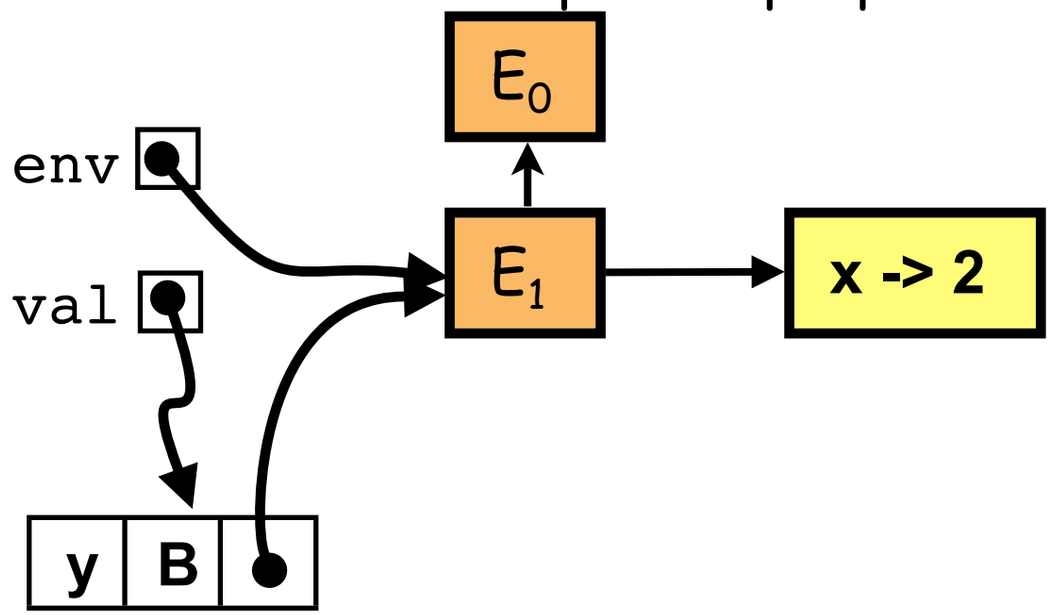
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = env.BeginScope();
env.Assoc("x", 2);
val = closure(n, B, env);
```



# Ambiente Mutável (revisitado)

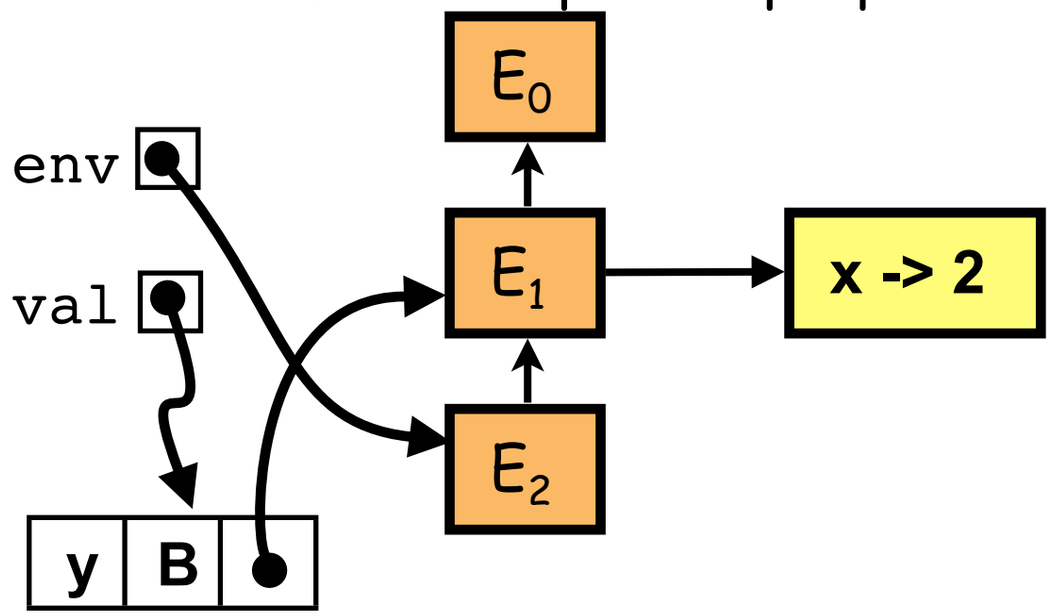
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = env.BeginScope();
env.Assoc("x", 2);
val = closure(n, B, env);
env.BeginScope();
```



# Ambiente Mutável (revisitado)

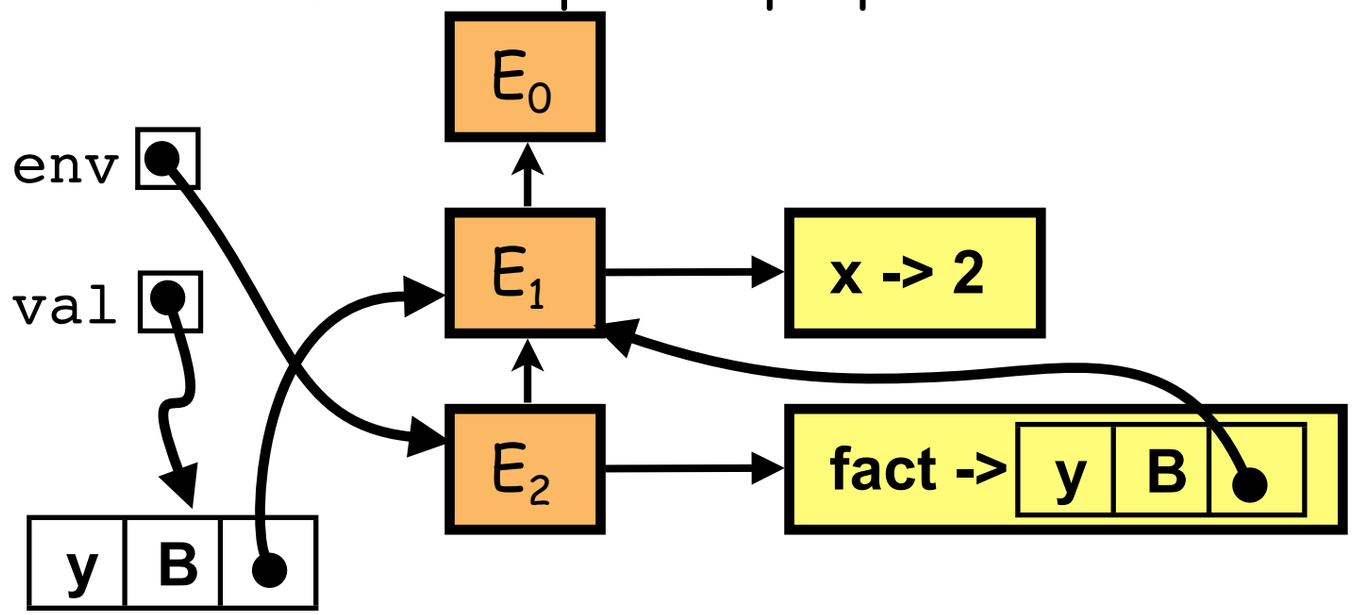
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = env.BeginScope();
env.Assoc("x", 2);
val = closure(n, B, env);
env.BeginScope();
env.Assoc("fact", val);
```



# Ambiente Mutável (revisitado)

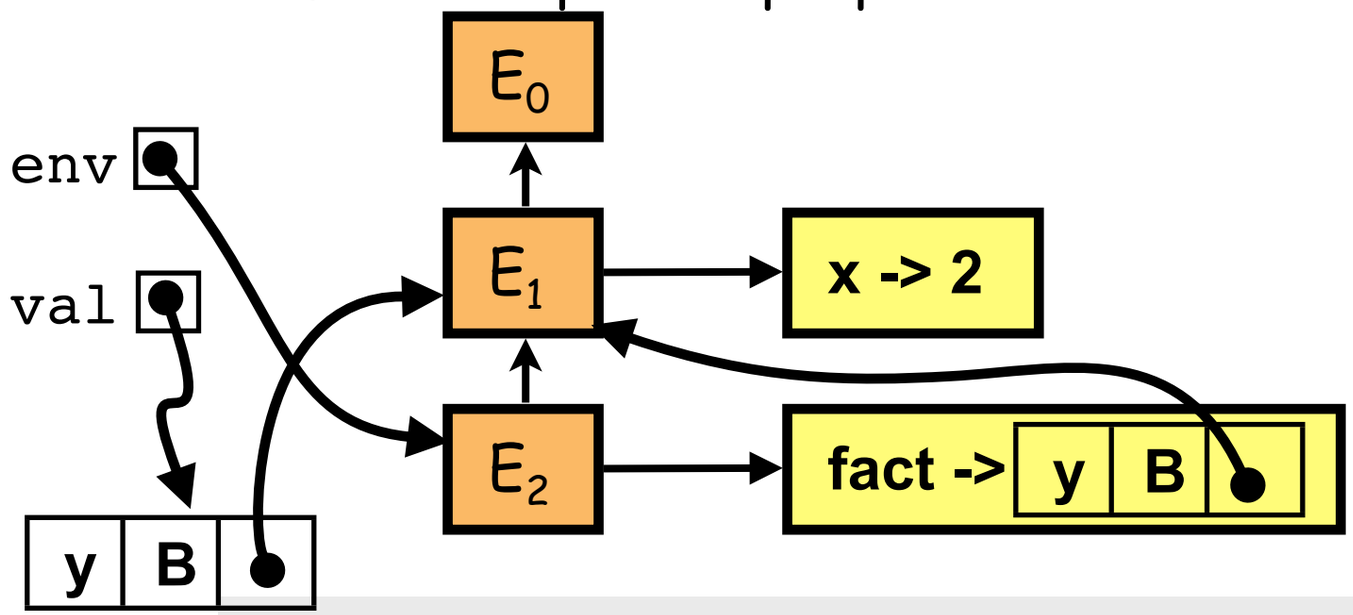
- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.

- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = env.BeginScope();
env.Assoc("x", 2);
val = closure(n, B, env);
env.BeginScope();
env.Assoc("fact", val);
```




Aqui, durante a avaliação do corpo B, o ambiente não refere a definição de fact (apenas de x) ...

# Semântica de RECF

- A função semântica I de RECF:

$$I : RECF \times ENV \rightarrow RESULT$$

RECF = conjunto dos programas abertos

ENV = conjunto dos ambientes válidos

RESULT = conjunto dos significados

- Os resultados podem ser valores inteiros, fechos (uma abstracção + um ambiente), ou um erro.

$$RESULT = integer \cup closure \cup \{ error \}$$

# Semântica de RECF

- Algoritmo eval para calcular a denotação de qualquer expressão E de RECF num ambiente env:

$$eval : RECF \times ENV \rightarrow RESULT$$

Dado um ambiente **env**

Se E é da forma **if**(E1, E2 , E3):
```
c = eval(E1, env);
if c != 0 then eval(E, env) ≜ eval(E2, env)
else eval(E, env) ≜ eval(E3, env)
```

Se E é da forma **declrec**(id, E1, E2):
```
[ envloc = env.BeginScope();
  envloc.Assoc(id, null);
  val = eval(E1, envloc);
  envloc.Update(id, val);
  v = eval(E2, envloc);
  envloc.EndScope() ]

eval(E, env) ≜ v
```

# Auto-avaliação...

- Avalie o programa P escrito na linguagem **RECF** usando a semântica apresentada:

```
declrec
   fact = fun n → if n then 1 else n*fact(n-1) end
in
   fact(3)
end
```

E1 = [ fact → clos(n, (if …) , $E_1$) ]

eval(P, ∅) =

eval(fact(3), E1) =

… ?

# Passagem de parâmetros

# Passagem de parâmetros

- Recorde a semântica da chamada de função adoptada nas linguagens CALCF e RECF:

Se E é da forma **call($E_1$, $E_2$)** :
**if closure(***id***,** *envc*, **B)** = **eval**($E_1$, *env*)
**then [** *envloc* = *envc*.BeginScope();
*envloc*.Assoc(*id*, **eval**($E_2$, *env*));
**eval**(E,*env*) ≜ **eval**(**B**, *envloc*) **]**
**else eval**(E,*env*) ≜ error

- A expressão E2 que denota o argumento da função é avaliada antes da função denotada pela expressão E1 ser efectivamente aplicada.

- Qual o valor intuitivo/efectivo do programa seguinte?

```
declrec f = fun x → f(x) end in
    decl g = fun y → 1 end in g(f(1)) end end
```

# Passagem de parâmetros por valor

- Qual o valor da expressão seguinte ?

```
declrec f = fun x → f(x) end in
   decl g = fun y → 1 end in g(f(1)) end end
```

- Valor de acordo com a semântica "declarativa":
  - `g` é a função constante que devolve sempre o valor `1` independentemente do valor do argumento.
  - Então, o valor `g(f(1))` deverá ser `1`.
- Valor determinado pela semântica "operacional":
  - A semântica apresentada não define valor para esta expressão, pois a avaliação de `f(1)` não termina!
  - A regra de passagem de argumentos utilizada é a "passagem por valor" (*call-by-value*)

# Passagem de parâmetros por "nome"

- Qual o valor da expressão (1 ou nenhum)?

```
declrec f = fun x → f(x) end in
    decl g = fun y → 1 end in g(f(1)) end end
```

- Existe um modo de avaliação que permite sempre encontrar o valor "declarativo" das expressões.

- Consiste em suspender a avaliação dos argumentos das funções até ao momento em que estes se tornam necessários.

- Corresponde em passar a expressão como argumento, em vez do seu resultado.

- A esta regra de avaliação de argumentos chama-se "passagem por nome" (*call-by-name*) (CBN)

- A estratégia de avaliação por omissão do Algol 60.

# Resultados de RECF (CBN)

- Algoritmo eval para calcular a denotação de qualquer expressão E de RECF num ambiente env usando a passagem de parâmetros CBN:

$$eval : RECF \times ENV \rightarrow RESULT$$

- Construtores do tipo de dados RESULT

```
num:      integer → RESULT
closure:  string × RECF × ENV → RESULT
susp:     RECF × ENV → RESULT
```

- Uma suspensão representa uma expressão por avaliar, juntamente com o ambiente respectivo.
- É necessário guardar na suspensão, junto com cada expressão, o ambiente correcto, pois em geral a expressão contém identificadores livres.

# Semântica de RECF (CBN)

- Algoritmo eval para calcular a denotação de qualquer expressão E de RECF num ambiente env usando a passagem de parâmetros CBN:

$$\text{eval} : \text{RECF} \times \text{ENV} \rightarrow \text{RESULT}$$

Se E é da forma **id**(s): val = *env*.Find(s)
**if** val = **susp**(B, *envsusp*)
**then eval**(E, *env*) ≜ **eval**(B, *envsusp*)
**else eval**(E, *env*) ≜ val

Se E é da forma **fun**(id, B): **eval**(E, *env*) ≜ **closure**(ident, B, *env*)

Se E é da forma **call**(E1, E2): **if eval**(E1, env) = **closure**(id, B , *envloc*)
**then [** *envloc = envloc*.BeginScope();
*envloc*.Assoc(id, **susp**(E2, env));
**eval**(E, *env*) ≜ **eval**(B, *envloc*) **]**
**else eval**(E, *env*) ≜ error

# Semântica de RECF (CBN)

- **Qual o valor (CBN) da expressão P ?**

```
declrec f = fun x → f(x) end in
   decl g = fun y → 1 end in g(f(1)) end end
```

$A_1$ =[ f → closure(x, f(x), $A_1$)]

```
eval(P, ∅) =
eval(decl g = (fun y → 1) in g(f(1)), A1) =
eval(g(f(1)), [g → closure(y,1,A1), f → closure(x, f(x), A1)] ) =
eval(1, [y → susp(f(1),A2), g → closure(y,1,A1), f → ... ] ) = 1
```

$A_2$

# Avaliação RECF (CBN)

- Qual o valor (CBN) da expressão P ?

```
decl y = 2 in decl f = (fun z → z+y) in f(2+y)
```

```
eval(P, ∅) =
eval(decl f = (fun z → z+y) in f(2+y), [y → 2] ) =
eval(f(2+y), [f → closure(z, z+y, A1), y → 2] ) =
eval(z+y, [z → susp(2+y, A2), f → closure(z, z+y, A1), y → 2] ) =
eval(z, [z → susp(2+y, A2),f → closure(z, z+y, A1), y → 2] )+2 =
eval(2+y, A2 )+2 = 6
```

# Passagem de parâmetros por "necessidade"

- Qual o valor da expressão?

```
decl x = var(0) in
  declrec f = fun x → x := !x+1 end in
    decl g = fun y → y+y+!x end in g(f(x)) end end
```

- Se se suspender a avaliação dos argumentos das funções até ao momento em que estes se tornam necessários passando a expressão como argumento, em vez do seu resultado, a utilização repetida dos parâmetros pode levar a resultados não esperados.

- Para avaliar só uma vez cada suspensão é preciso "guardar" o seu resultado após a primeira avaliação.

- A esta regra de avaliação de argumentos chama-se "passagem por necessidade" (*call-by-need*). Utilizada em linguagens com avaliação Lazy (e.g. Haskell)

# Semântica de RECF (Lazy)

- Algoritmo eval para calcular a denotação de qualquer expressão E de RECF num ambiente env usando a passagem de parâmetros Lazy:

$$eval : RECF \times ENV \rightarrow RESULT$$

- É necessário representar a suspensão por um valor mutável:

Se E é da forma **var**(id): val = *env*.Find(id)
**if** val = **susp** (B, *envsusp*)
**then [ if** val.isUnevaluated()
**then** val.set(**eval**(B, *envsusp*))
**eval**(E, *env*) ≜ val.getValue() **]**
**else eval**(E, *env*) ≜ val