Interpretação e Compilação (de Linguagens de Programação)

Unidade 6: 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.

```
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

- 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): $\lambda x.x$

- 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)))
```

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:

$$eval : CALCF \rightarrow RESULT$$

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

```
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)
```

```
    (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.

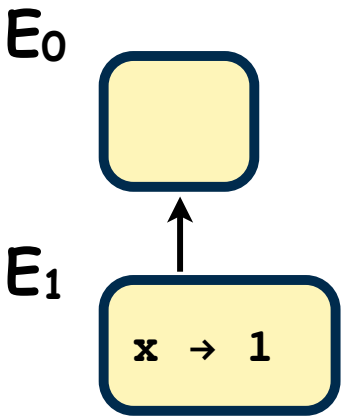
```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

Exemplo

E_0

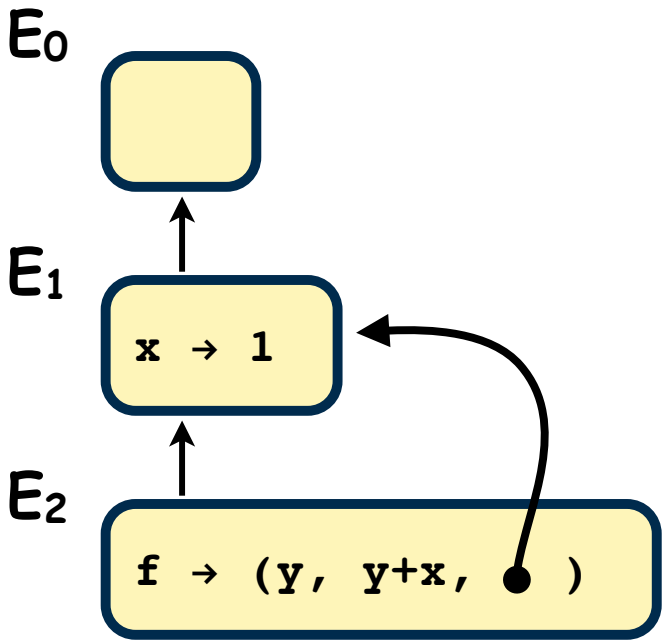
```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

Exemplo



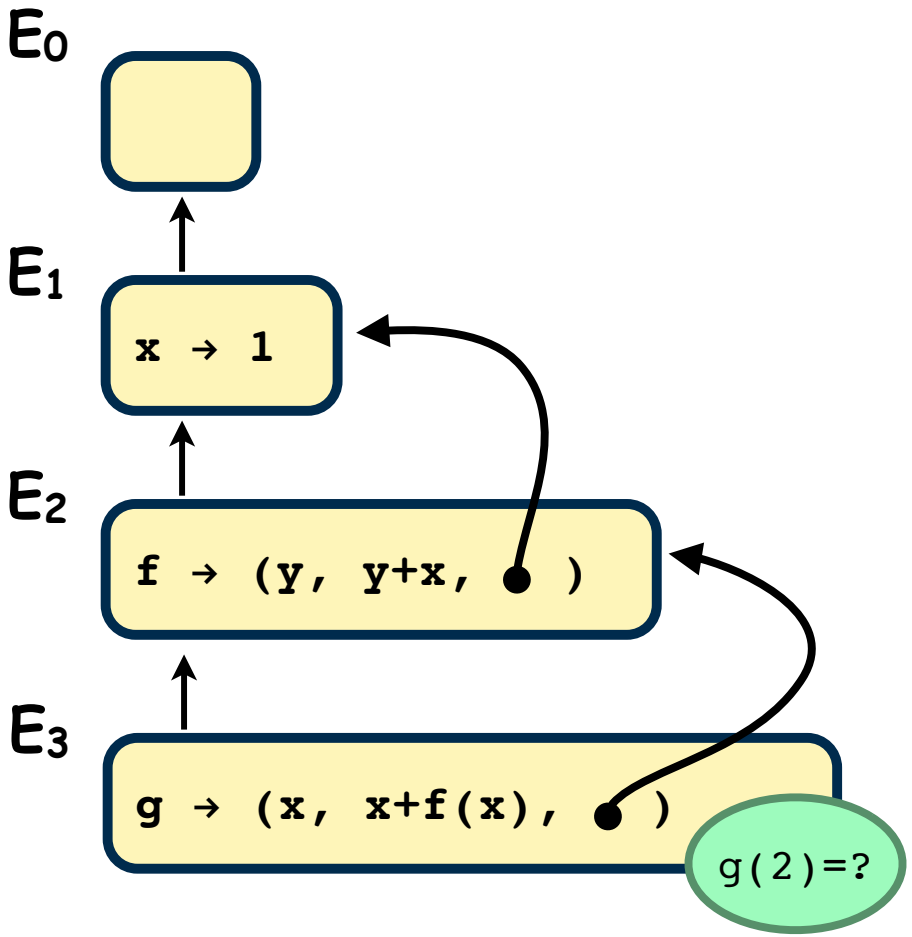

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

Exemplo



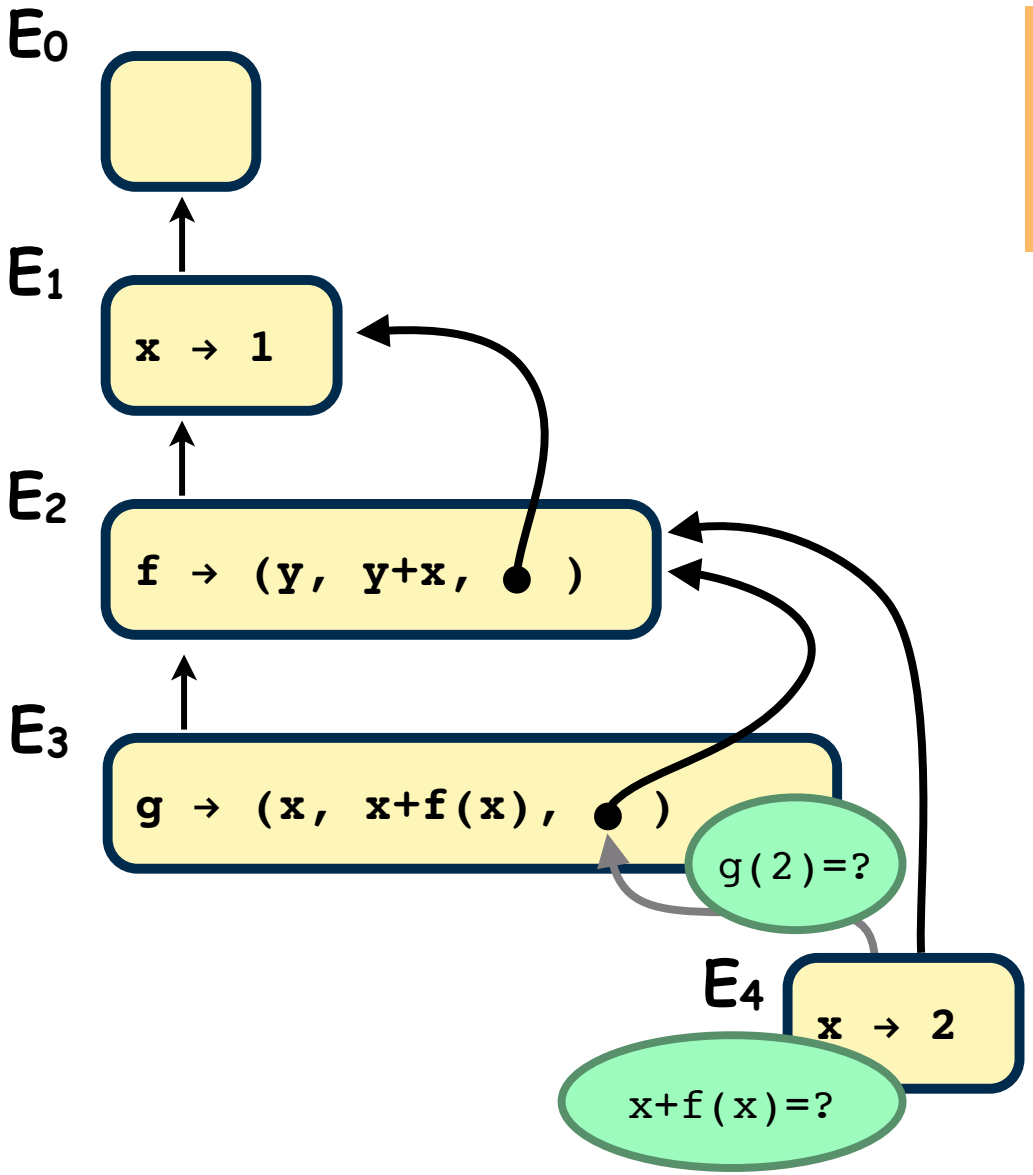

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

Exemplo




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

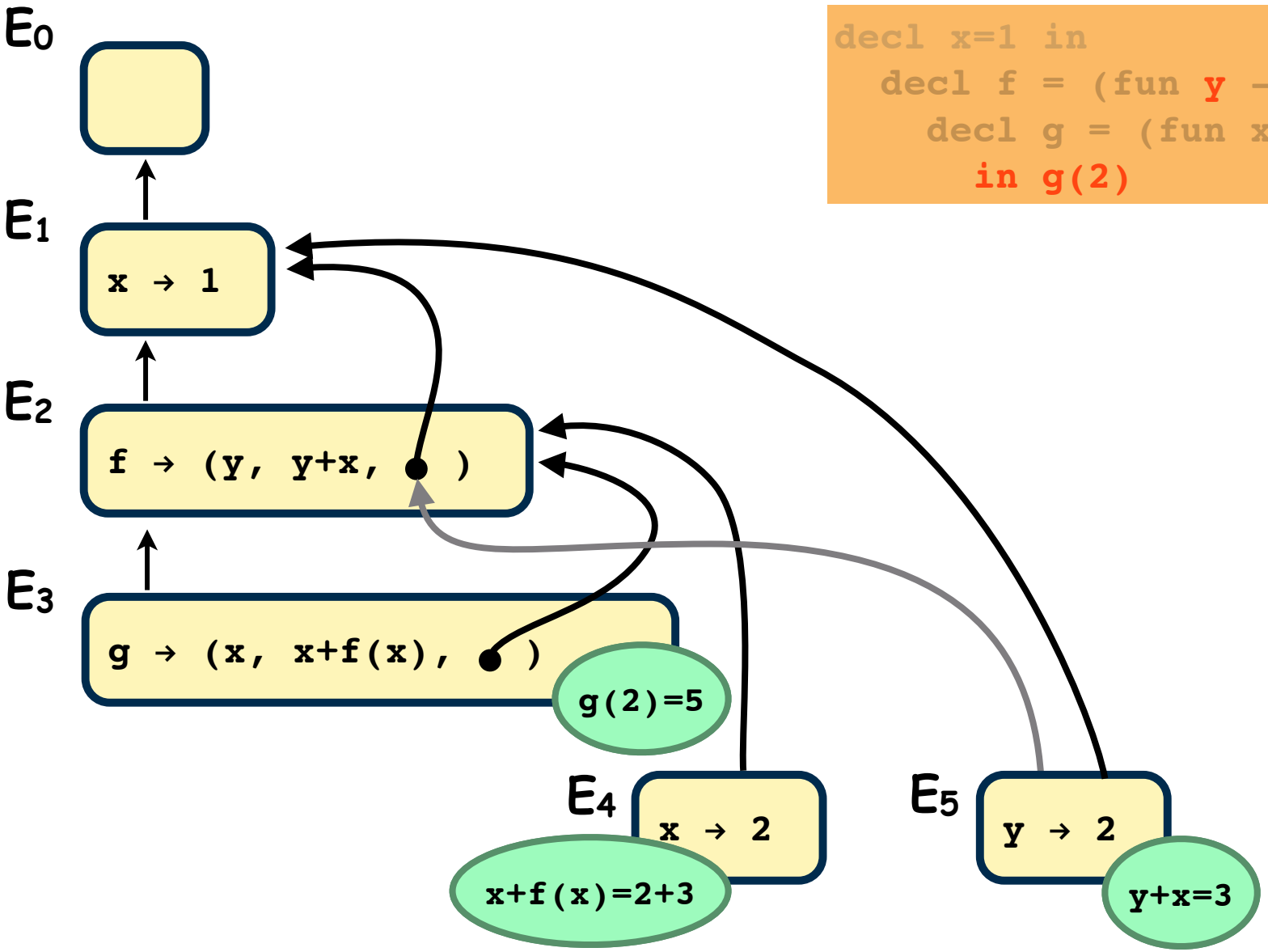
Exemplo




```
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)
```

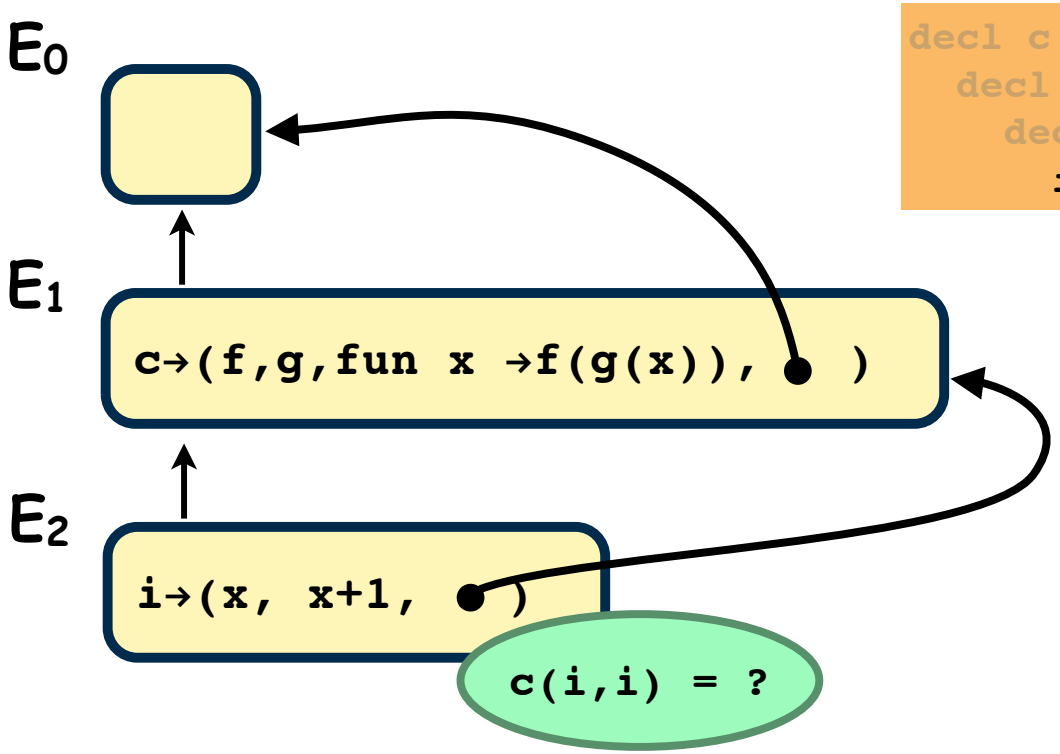


Avaliação de Funções

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

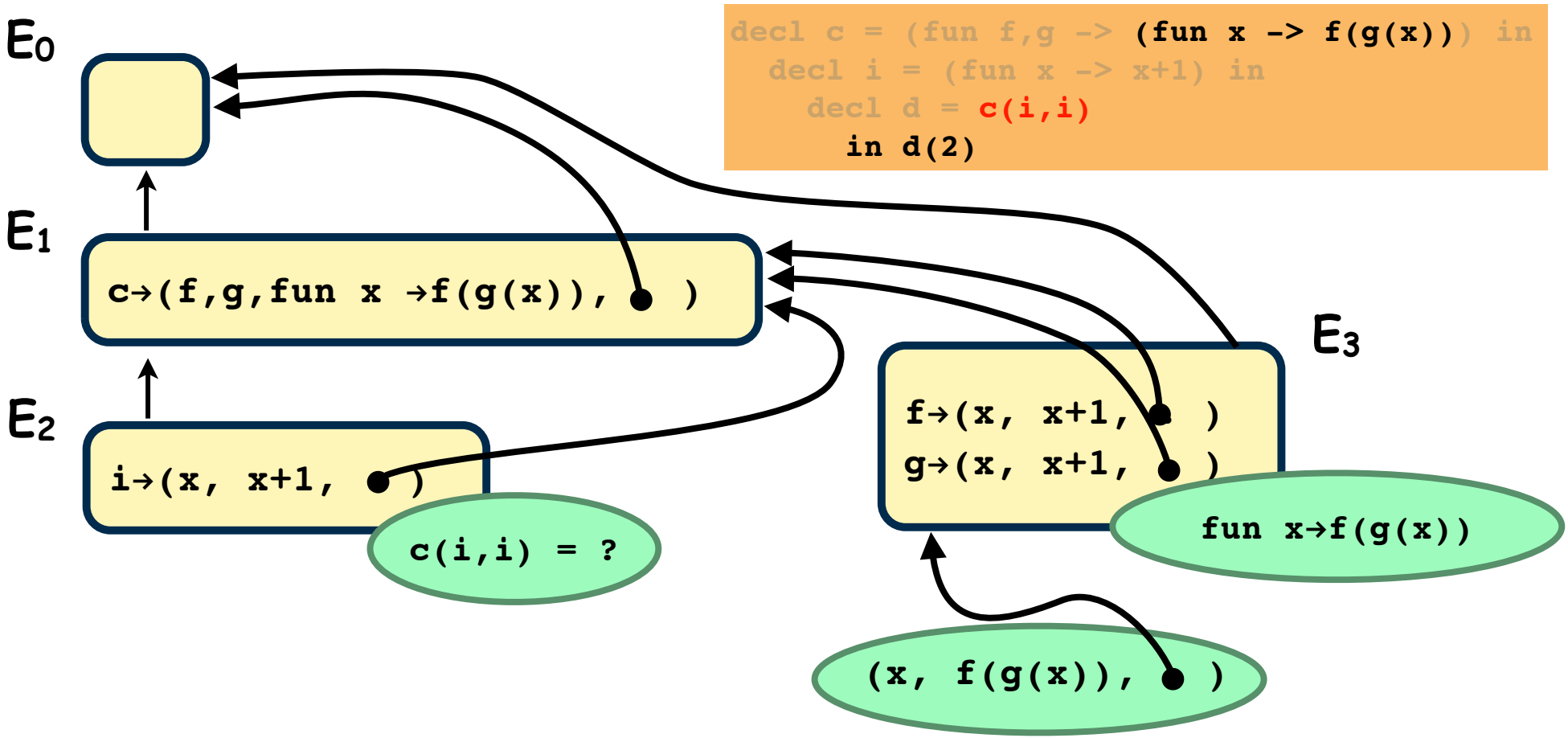
```
decl comp = (fun f,g -> (fun x -> f(g(x)))) in
  decl inc = (fun x -> x+1) in
    decl dup = comp(inc,inc)
      in dup(2)
```

Avaliação de Funções

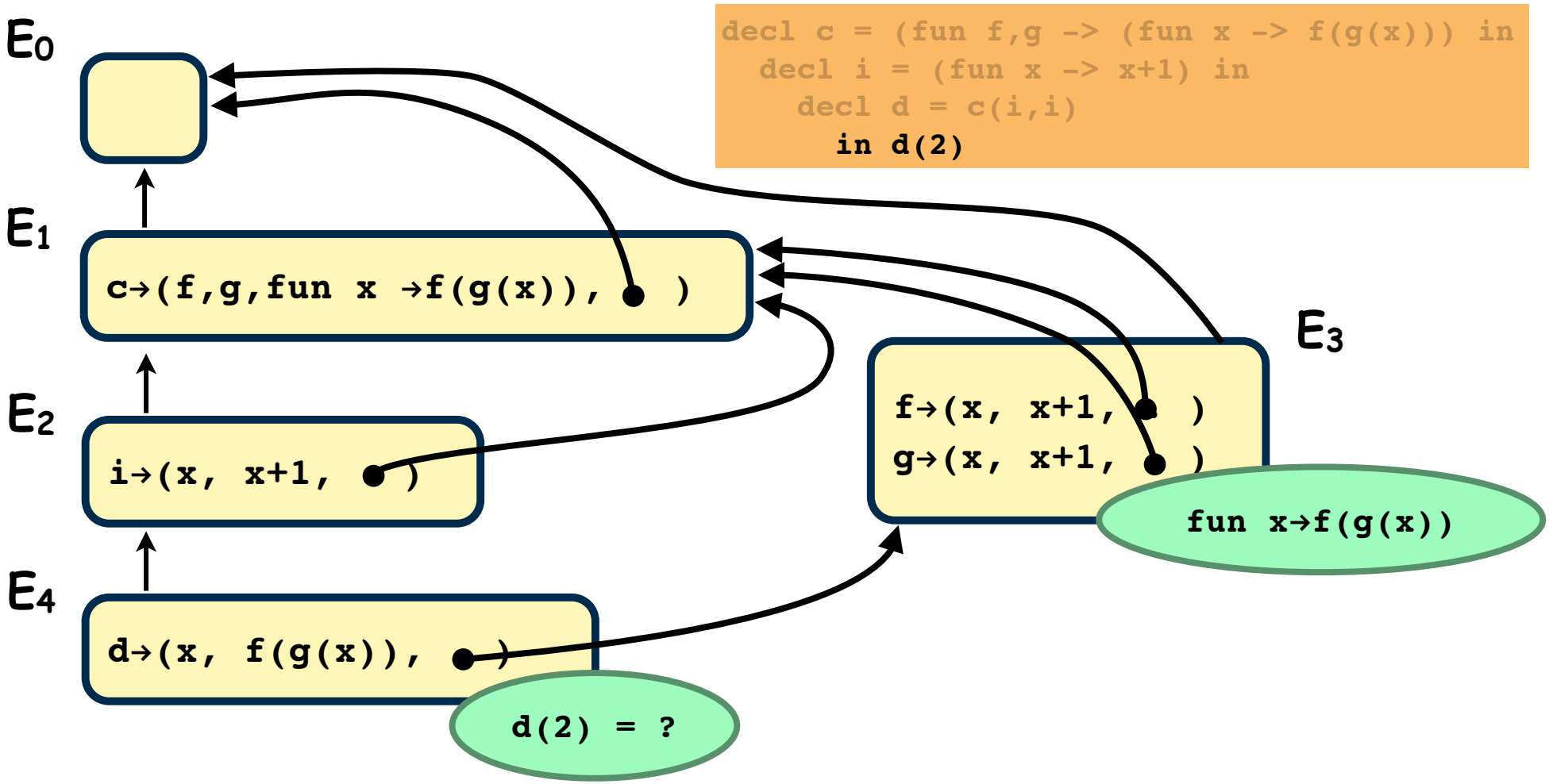



```
decl c = (fun f,g -> (fun x -> f(g(x))) in
  decl i = (fun x -> x+1) in
    decl d = c(i,i)
      in d(2)
```

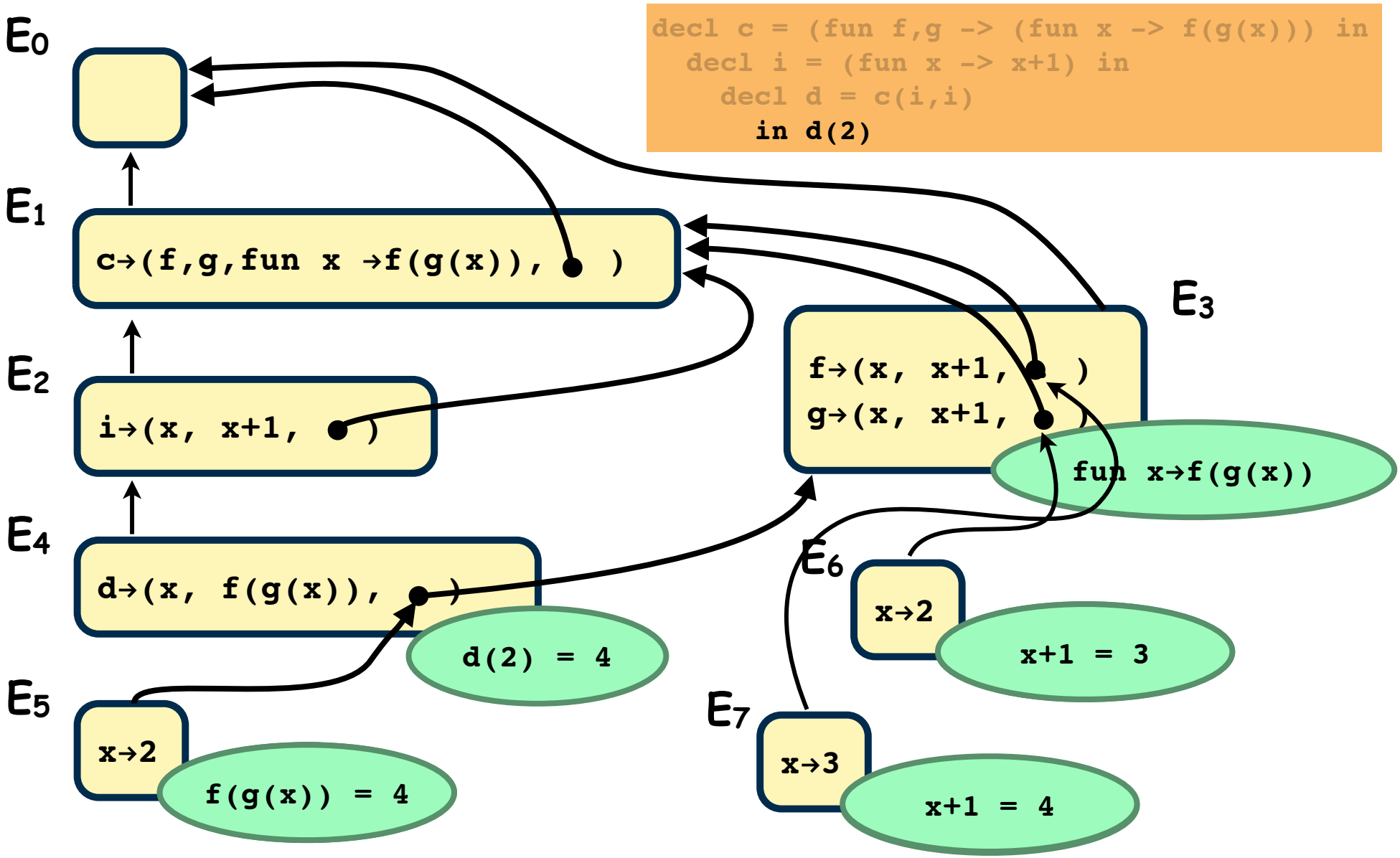
Avaliação de Funções



Avaliação de Funções



Avaliação de Funções



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"