

# Interpretação e Compilação de Linguagens de Programação

Mestrado Integrado em Engenharia Informática  
Departamento de Informática  
Faculdade de Ciências e Tecnologia  
Universidade Nova de Lisboa

2016-2017

João Costa Seco (joao.seco@di.fct.unl.pt)

# Lecture 04

## Ligação e âmbito

# Ligação e Âmbito

Os identificadores são a primeira ferramenta básica para criar abstrações numa linguagem de programação. Um identificador usado numa expressão (ou programa) representa uma subexpressão cuja definição é feita

- Literais e identificadores
- Declaração de identificadores
- Âmbito de uma declaração
- Ocorrências de um identificador (livres, ligadas e ligantes)
- Expressões abertas e fechadas
- Construção fundamental **decl** id=E **in** E **end**.
- Linguagem com identificadores (declaração e ocorrências ligadas): CALCI.
- Algoritmo interpretador com substituição
- Algoritmo interpretador com ambiente
- Algoritmo compilador com ambiente

# Constantes e Identificadores

- Constantes (ou literais)
  - Referem entidades ou valores bem determinados em **qualquer contexto** onde ocorram
  - Nas linguagens “naturais” correspondem aos “nomes próprios”.
  - Linguagem ML: **true, false, []**
  - Linguagem C: **1, 1.0, 0xFF, “hello”, int**
- Identificadores (ou nomes)
  - Referem entidades ou valores que **dependem do contexto**
  - Nas linguagens “naturais” correspondem aos “pronomes”.
  - Linguagem Java: **x, Count, System.out**
  - Linguagem C: **printf**

# Ligação e Âmbito

- Os **literais** e os **identificadores** denotam sempre uma entidade bem determinada e inalterável.
- A entidade denotada por um literal (ou o valor de um literal) é determinada pelo próprio literal (**23**, **"hi!"**, etc).
- A associação entre um identificador e a entidade ou valor denotado chama-se **ligação** (*binding*)
- Em geral, a ligação entre identificador e entidade denotada estabelece-se num certo contexto sintáctico e é introduzida por uma **declaração**
- Ao contexto sintáctico onde uma ligação tem efeito chama-se o **âmbito** (*scope*) da ligação

# Ligação e Âmbito

- O identificador **x** denota uma variável de estado (célula de memória)

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

# Ligação e Âmbito

- O identificador **x** denota uma variável de estado (célula de memória)

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

Âmbito da ligação

# Ligação e Âmbito

- O identificador **j** denota uma variável de estado (célula de memória)

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

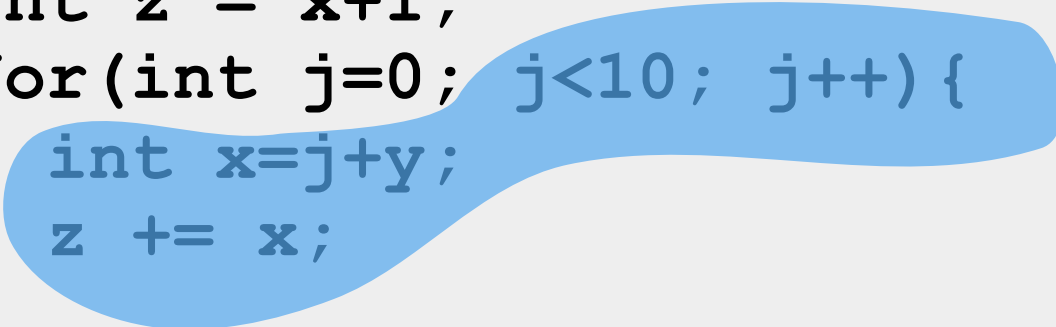


# Ligação e Âmbito

- O identificador **j** denota uma variável de estado (célula de memória)

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

Âmbito da ligação



# Elementos de um âmbito

- A **ligação** entre um identificador e a respectiva entidade por este denotada (valor, posição de memória, etc) envolve os seguintes ingredientes:
- Uma (única!) **ocorrência ligante**:  
em geral, corresponde à declaração do identificador.
- O **âmbito** da ligação  
A “parte/região/zona/fragmento” do programa onde a ligação em causa tem efeito
- Várias (zero ou mais) ocorrências **ligadas**  
todas as ocorrências do identificador, distintas da ocorrência ligante, que existem dentro do âmbito

# Ocorrências ligantes e ligadas

- Ocorrências do identificador **x**

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

Ocorrências ligantes

# Ocorrências ligantes e ligadas

- Ocorrências do identificador **x**

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

Ocorrências ligadas

# Ocorrências ligadas

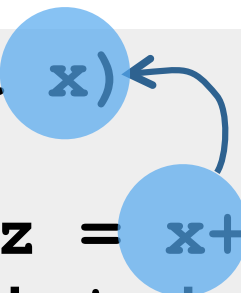
- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

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

# Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

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



# Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

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

# Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

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



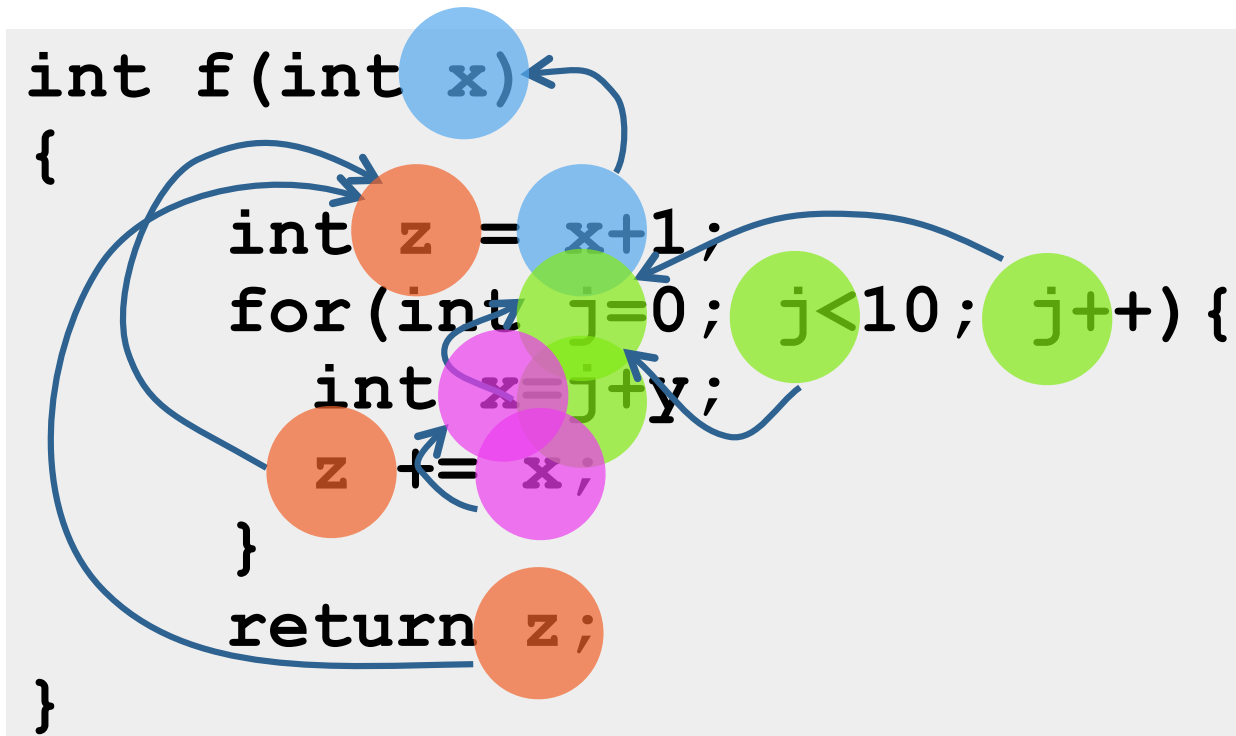
# Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

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

# Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)



# Ocorrências livres

- Uma ocorrência de identificador que não é ligada nem ligante diz-se **livre**

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

# Expressões Abertas e Fechadas

- Uma subexpressão diz-se **aberta** se contém ocorrências livres de identificadores
- Uma subexpressão diz-se **fechada** se não contém ocorrências livres de identificadores
- Exemplos de expressões abertas:

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


```
let x=1 in (f x)                                   OCaml
```

# Expressões Abertas e Fechadas

- Uma subexpressão diz-se **aberta** se contém ocorrências livres de identificadores
- Uma subexpressão diz-se **fechada** se não contém ocorrências livres de identificadores
- Exemplos de expressões abertas:

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

ocorrência livre **C**



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

ocorrência livre **OCaml**



# Semântica de expressões abertas

- A denotação de uma subexpressão de programa só pode ser calculada se se conhecer a denotação de cada identificador que nela ocorra livre.
- A definição de uma semântica composicional para linguagens com declarações de identificadores tem necessariamente que considerar expressões abertas.

Por exemplo, a expressão OCaml

```
let x = 2 in (x+x)
```

é fechada mas contém uma subexpressão aberta.

- Um programa (fragmento fechado) pode conter no seu interior expressões abertas.

[Dê exemplos de linguagens de programação onde seja possível compilar um programa aberto]

# Ambiente

- Um programa fechado fornece necessariamente ligações para todas as ocorrências livres de identificadores que ocorram nas suas subexpressões (através de declarações).

Para cada subexpressão  $E$  de um programa  $P$ , ao conjunto de todas as ligações no âmbito das quais  $E$  ocorre chama-se o **ambiente** de  $E$  em  $P$ .

# Ambiente (Quiz)

- Qual o ambiente da subexpressão “**x+1**”?

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



# Ambiente (Quiz)

- Qual o ambiente da subexpressão “**z+=x**”?

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

# Ambiente (Quiz)

- Qual o ambiente da subexpressão “**return z**”?

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

# Exemplo: A Linguagem CALCI

- A linguagem CALCI estende a linguagem CALC com a possibilidade de se poderem introduzir e usar identificadores usando a construção **declare**:

```
decl Id = Expressão1 in Expressão2 end
```

Numa expressão **decl**, a primeira ocorrência de *Id* é **ligante**, no âmbito definido pela *Expressão2*

- Definimos os programas CALCI como sendo as **expressões fechadas** da linguagem CALCI.

```
decl x=2 in decl y=x+2 in (x+y) end end
```

# A Linguagem CALCI (como tipo indutivo)

- Tipo de dados CALCI com os construtores: num, add, mul, div, sub, id, decl

**num:** Integer  $\rightarrow$  CALCI  
**id:** String  $\rightarrow$  CALCI  
**add:** CALCI  $\times$  CALCI  $\rightarrow$  CALCI  
**mul:** CALCI  $\times$  CALCI  $\rightarrow$  CALCI  
**div:** CALCI  $\times$  CALCI  $\rightarrow$  CALCI  
**sub:** CALCI  $\times$  CALCI  $\rightarrow$  CALCI  
**decl:** String  $\times$  CALCI  $\times$  CALCI  $\rightarrow$  CALCI

# A Linguagem CALCI (como tipo indutivo)

- Tipo de dados CALCI com os construtores: num, add, mul, div, sub, id, decl

**num:** Integer  $\rightarrow$  CALCI

**id:** String  $\rightarrow$  CALCI

**add:** CALCI  $\times$  CALCI  $\rightarrow$  CALCI

**mul:** CALCI  $\times$  CALCI  $\rightarrow$

**div:** CALCI  $\times$  CALCI  $\rightarrow$

**sub:** CALCI  $\times$  CALCI  $\rightarrow$

**decl:** String  $\times$  CALCI  $\times$  C

```
type calci =  
  | Number of int  
  | Add of calci * calci  
  | Sub of calci * calci  
  | Mul of calci * calci  
  | Div of calci * calci  
  
  | Id of string  
  | Decl of string * calci * calci
```

```
  | Decl of string * calci * calci  
  | Id of string
```

# Semântica de CALCI (1)

- A função semântica  $I$  de CALCI pode ser definida por um **algoritmo** que “sabe como interpretar” **todas** as expressões de CALCI, determinando o seu **valor ou efeito**.

$$I : \text{CALCI} \rightarrow \text{Integer}$$

$\text{CALCI}$  = conjunto das expressões fechadas

$\text{Integer}$  = conjunto dos significados (denotações)

# Interpretador de CALCI

- Algoritmo  $\text{eval}(E)$  para calcular o valor de uma expressão fechada qualquer  $E$  de CALCI:

$\text{eval} : \text{CALCI} \rightarrow \text{Integer}$

$\text{eval}(\text{ num}(n) )$	$\triangleq n$
$\text{eval}(\text{ add}(E1, E2) )$	$\triangleq \text{eval}(E1) + \text{eval}(E2)$
...	
$\text{eval}(\text{ decl}(s, E1, E2) )$	$\triangleq ???$

**Intuitivamente:** o significado da expressão contendo identificadores deve ser o mesmo da expressão em que os identificadores são substituídos pelas subexpressões que eles representam.

# Interpretador de CALCI

- Algoritmo  $\text{eval}(E)$  para calcular o valor de uma expressão fechada qualquer  $E$  de CALCI:

$\text{eval} : \text{CALCI} \rightarrow \text{Integer}$

$\text{eval}(\text{num}(n))$	$\triangleq n$
$\text{eval}(\text{add}(E1, E2))$	$\triangleq \text{eval}(E1) + \text{eval}(E2)$
...	
$\text{eval}(\text{decl}(s, E1, E2))$	$\triangleq [ G = \text{subst}(E1, s, E2); \text{eval}(G); ]$

**Intuitivamente:** o significado da expressão contendo identificadores deve ser o mesmo da expressão em que os identificadores são substituídos pelas subexpressões que eles representam.



# A Função Subst

$\text{subst}( F, s, E )$

Calcula a expressão que resulta de substituir todas as ocorrências livres do identificador  $s$  pela expressão  $F$  na expressão  $E$ .

$\text{subst}(y+z, s, s+s+2) = (y+z)+(y+z)+2$

$\text{subst}(u, y, \text{decl } x=y \text{ in decl } y=2 \text{ in } x+y) = \text{decl } x=u \text{ in decl } y=2 \text{ in } x+y$

# Definição da Função Subst

**subst**(F, s, **num**(n) )  $\triangleq$  num(n);

**subst**(F, s, **id**(s) )  $\triangleq$  F;

**subst**(F, s, **add**(E1, E2) )  $\triangleq$  add( **subst**(F, s, E1), **subst**(F, s, E2));

...

**subst**(F, s, **decl**(s, E1, E2))  $\triangleq$  [/ \* caso s = s' \*/ \* ???]

**subst**(F, s, **decl**(s', E1, E2))  $\triangleq$  [/ \* caso s  $\neq$  s' \*/ ??? ]

# Definição da Função Subst

**subst**(F, s, **num**(n) )  $\triangleq$  **num**(n);

**subst**(F, s, **id**(s) )  $\triangleq$  F;

**subst**(F, s, **add**(E1, E2) )  $\triangleq$  **add**( **subst**(F, s, E1), **subst**(F, s, E2));

...

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

**subst**(F, s, **decl**(s', E1, E2))  $\triangleq$  [**\*** caso  $s \neq s'$  **\*** / G = **subst**(F, s, E1);  
                                  **decl**(s', G, **subst**(F, s, E2)); ]

# Definição da Função Subst

**subst(s, num(*n*), F )**       $\triangleq$  num(*n*);

**subst(s, id(s), F )**       $\triangleq$  F;

**subst(s, add(E1, E2) )**

...

**subst(s, decl(s, E1, E2),**

**subst(s, decl(s', E1, E2),**

```
let rec subst e x e' =  
  let subst' = subst e x in  
  match e' with  
  | Number n -> e'  
  | Add (l,r) -> Add(subst' l, subst' r)  
  | Sub (l,r) -> Sub(subst' l, subst' r)  
  | Mul (l,r) -> Mul(subst' l, subst' r)  
  | Div (l,r) -> Div(subst' l, subst' r)  
  | Decl (y,l,r) ->  
    if x = y then Decl(y,subst' l,r)  
    else Decl(y,subst' l, subst' r)  
  | Id y -> if x = y then e else e'
```

```
| Id λ -> if x = λ then fmem e e' else  
  e' subst' (λ',subst' f, λ')  
  if x = λ then fmem subst' (λ',subst' f, λ')
```

# Interpretador de CALCI

- Algoritmo  $\text{eval}(E)$  para calcular o valor de uma expressão fechada qualquer  $E$  de CALCI:

$\text{eval} : \text{CALCI} \rightarrow \text{integer}$

<b>eval( num(<math>n</math>) )</b>	$\triangleq n$
<b>eval( add(<math>E_1, E_2</math>) )</b>	$\triangleq \text{eval}(E_1) + \text{eval}(E_2)$
...	
<b>eval( decl(<math>s, E_1, E_2</math>) )</b>	$\triangleq \text{eval}(\text{subst}(E_1, s, E_2)); ]$

# Interpretador de CALCI

- Algoritmo  $\text{eval}(E)$  para calcular o valor de uma expressão fechada qualquer  $E$  de CALCI:

$\text{eval} : \text{CALCI} \rightarrow \text{integer}$

$\text{eval}(\text{num}(n)) \triangleq n$

```
eval( add l r ) =
  let rec eval a =
    match a with
    | Number n -> n
    | Add (l,r) -> (eval l) + (eval r)
    | Sub (l,r) -> (eval l) - (eval r)
    | Mul (l,r) -> (eval l) * (eval r)
    | Div (l,r) -> (eval l) / (eval r)
    | Decl (s,l,r) -> eval (subst l s r)
    | Id s -> ???
```

```
| Id s -> ???
| Decl (s,l,r) -> eval (subst l s r)
| Decl (s,l,r) -> eval (subst l s r)
```

# Interpretador de CALCI

- Algoritmo  $\text{eval}(E)$  para calcular o valor de uma expressão fechada qualquer  $E$  de CALCI:

$\text{eval} : \text{CALCI} \rightarrow \text{integer}$

$\text{eval}(\text{num}(n)) \triangleq n$

```
eval( add let rec eval a =  
    match a with  
    | Number n -> n  
    | Add (l,r) -> (eval l) + (eval r)  
    | Sub (l,r) -> (eval l) - (eval r)  
    | Div (l,r) -> (eval l) / (eval r)  
    | Decl (s,l,r) -> eval (subst l s r)  
    | Id s -> ???  
    | Iq z -> ???  
    | DGCJ (z',l',r) -> eval (subst l' z' r)  
    | DGA (l,r) -> (eval l) \ (eval r)
```

é preciso programar o caso do identificador??

# Interpretador de CALCI

- Algoritmo  $\text{eval}(E)$  para calcular o valor de uma expressão fechada qualquer  $E$  de CALCI:

$\text{eval} : \text{CALCI} \rightarrow \text{integer}$

$\text{eval}(\text{num}(n)) \triangleq n$

$\text{eval}(\text{add } l, r) =$

$\text{match } a \text{ with}$

$| \text{Number } n \rightarrow n$

$| \text{Add } (l, r) \rightarrow (\text{eval } l) + (\text{eval } r)$

$| \text{Sub } (l, r) \rightarrow (\text{eval } l) - (\text{eval } r)$

Porquê? o que fazer?

$| \text{Div } (l, r) \rightarrow (\text{eval } l) / (\text{eval } r)$

$| \text{Decl } (s, l, r) \rightarrow \text{eval } (\text{subst } l \text{ s } r)$

$| \text{Id } s \rightarrow ???$

$| \text{Eq } e \rightarrow ???$

$| \text{DGC } (s, l, r) \rightarrow \text{eval } (\text{subst } l \text{ s } r)$

$| \text{DGA } (s, l, r) \rightarrow (\text{eval } l) \setminus (\text{eval } r)$



# Interpretador de CALCI

- Algoritmo  $\text{eval}(E)$  para calcular o valor de uma expressão fechada qualquer  $E$  de CALCI:

$\text{eval} : \text{CALCI} \rightarrow \text{integer}$

$\text{eval}(\text{num}(n)) \triangleq n$

```
eval( add l r ) =  
  let rec eval a =  
    match a with  
    | Number n -> n  
    | Add (l,r) -> (eval l) + (eval r)  
    | Sub (l,r) -> (eval l) - (eval r)
```

Estamos a trabalhar com expressões fechadas. A existência de identificadores denota a falta de uma declaração, um erro!



# Semântica de CALCI (2)

- A semântica da linguagem CALCI baseada em substituições é muito conveniente do ponto de vista da especificação pois é muito simples.

$\text{eval} : \text{CALCI} \rightarrow \text{Integer}$

- Já do ponto de vista operacional, é conveniente definir uma semântica mais concreta, recorrendo à manipulação de ambientes.
- A manipulação de ambientes também é mais conveniente como técnica de implementação de interpretadores (como estruturas de dados auxiliares).

# Semântica de CALCI (2)

A função semântica  $I$  de CALCI pode ser definida por um **algoritmo** interpretador para expressões de CALCI, determinando o seu **valor ou efeito**, dado um ambiente contendo os valores dos seus identificadores livres.

$$I : \text{CALCI} \times \text{ENV} \rightarrow \text{Integer}$$

$\text{CALCI}$  = programas abertos

$\text{ENV}$  = ambientes

$\text{Integer}$  = significados (denotações)

# Semântica de CALCI (2)

A função semântica  $I$  de CALCI pode ser definida por um **algoritmo** interpretador para expressões de CALCI, determinando o seu **valor ou efeito**, dado um ambiente contendo os valores dos seus identificadores livres.

$I : \text{CALCI} \times \text{ENV} \rightarrow \text{integer}$

```
CALCI = let rec eval e env =  
      match e with  
ENV    =      | Number n -> n  
          | Id s -> find s env  
  
Integer =      | Add (l,r) -> (eval l) + (eval r)  
          | Sub (l,r) -> (eval l) - (eval r)  
          | Mul (l,r) -> (eval l) * (eval r)  
          | Div (l,r) -> (eval l) / (eval r)  
  
          | Decl (s,l,r) -> ...
```

# Semântica de CALCI (2)

A função semântica  $I$  de CALCI pode ser definida por um **algoritmo** interpretador para expressões de CALCI, determinando o seu **valor ou efeito**, dado um ambiente contendo os valores dos seus identificadores livres.

CALCI

ENV

Integer

$I : \text{CALCI} \times \text{ENV} \rightarrow \text{integer}$

```
let rec eval e env =  
  match e with  
    | Number n -> n  
    | Id s -> find s env  
  
    | Add (l,r) -> (eval l env) + (eval r env)  
    | ....  
  
    | Decl (s,l,r) ->  
      let v = eval l env in  
      let new_env = assoc s v env in  
      eval r new_env
```



# Ambiente “mutável”

- Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável ao estilo Object-Oriented.
- Numa linguagem estruturada, com âmbitos encaixados hierarquicamente, a adição e remoção de ligações entre identificadores e valores segue uma disciplina LIFO.
- Um ambiente guarda as associações correspondentes a um determinado âmbito (e todos os âmbitos envolventes). A partir de um ambiente pode criar-se um novo nível, correspondendo a um âmbito encaixado.

**Environ BeginScope()**

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

**Environ EndScope()**

- que coloca o ambiente no estado anterior à última operação BeginScope().



# Ambiente “mutável”

- Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável ao estilo Object-Oriented.
- Outras operações fundamentais:

**void** Assoc(String id, Value val)

- Adiciona uma nova ligação que associa ao identificador **id** o valor **val** indicado.
  - A ligação é adicionada ao último nível (mais recente) do ambiente.
- Devolve o valor associado ao identificador **id** no ambiente.

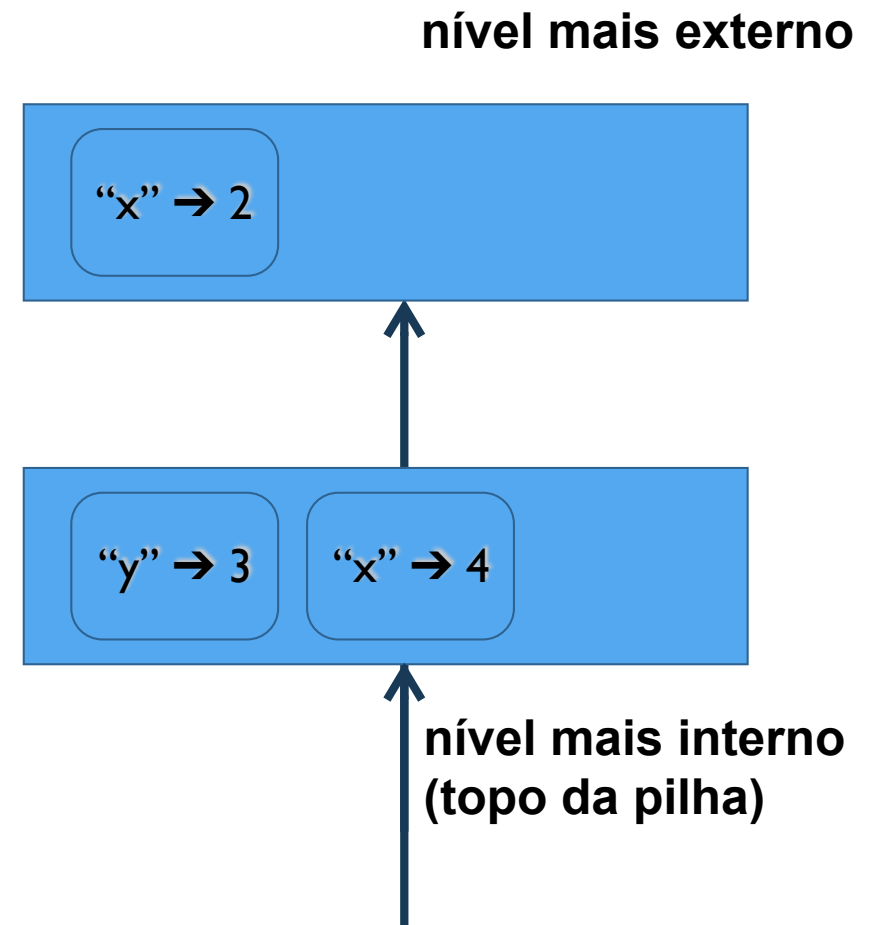
**Value** Find(String id)

- A pesquisa é efectuada do nível mais “recente” para o mais “antigo”, de modo a respeitar o encaixe dos âmbitos das declarações.

# A “interface” Ambiente

- Simule mentalmente:

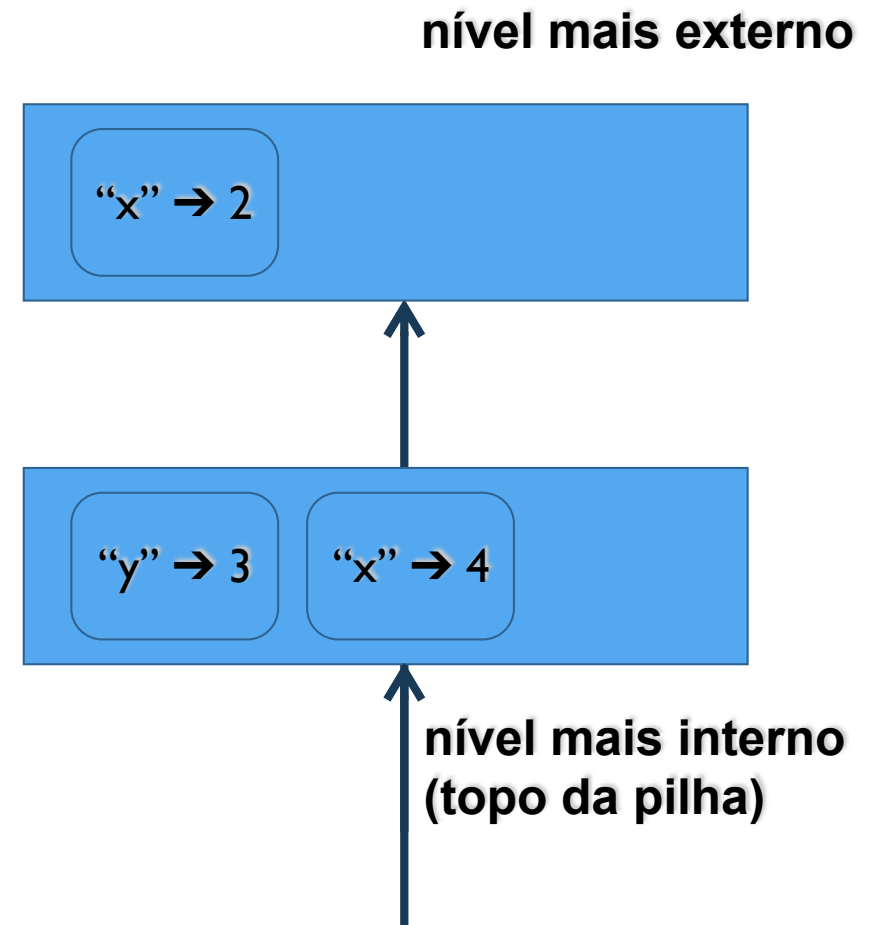
```
env = new Environment();  
env.Assoc("x", 2);  
val = env.Find("x");      // devolve 2  
env = env.BeginScope();  
env.Assoc("y", 3);  
env.Assoc("x", 4);  
val = env.Find("y");      // devolve 3  
val = env.Find("x");      // devolve 4  
env=env.EndScope()  
val = env.Find("x")       // devolve 2
```



# A “interface” Ambiente

- Implementado como pilha de dicionários ...

```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x");      // devolve 2
env = env.BeginScope();
env.Assoc("y", 3);
env.Assoc("x", 4);
val = env.Find("y");      // devolve 3
val = env.Find("x");      // devolve 4
env=env.EndScope()
val = env.Find("x")       // devolve 2
```



# Interpretador de CALCI

- Algoritmo  $\text{eval}(E, \text{env})$  para calcular o valor de uma expressão  $E$  da linguagem CALCI:

$\text{eval} : \text{CALCI} \times \text{ENV} \rightarrow \text{Integer}$

```
eval( num( $n$ ) ,  $\text{env}$ )       $\triangleq n$ 
eval( id( $s$ ) ,  $\text{env}$ )         $\triangleq \text{env.Find}(s)$ 
eval( add( $E1, E2$ ) ,  $\text{env}$ )   $\triangleq \text{eval}(E1, \text{env}) + \text{eval}(E2, \text{env})$ 
...
eval( decl( $s, E1, E2$ ),  $\text{env}$ )  $\triangleq$  [  $v1 = \text{eval}(E1, \text{env});$ 
                                    $\text{env} = \text{env.BeginScope}();$ 
                                    $\text{env.Assoc}(s, v1);$ 
                                    $\text{val} = \text{eval}(E2, \text{env});$ 
                                    $\text{env} = \text{env.EndScope}();$ 
                                    $\text{val}$  ]
```

# Erros de execução

- O que é que acontece se não for encontrado o identificador no ambiente?  
A que corresponde essa falha?  
A um erro de execução do tipo “identificador não declarado”.
- A função não está definida para os programas em que há ocorrências de identificadores sem declaração.
- Que outros tipos de erros de execução podem ocorrer?
- São o mesmo tipo de erros? É possível evitar uns e outros não?

# Compilação de Declarações (Compiler Frames)

# Compilador de CALCI

- Algoritmo  $\text{comp}(E)$  para traduzir uma expressão  $E$  qualquer de CALC numa sequência de instruções CLR

**$\text{comp} : \text{CALCI} \rightarrow \text{CodeSeq}$**

se $E$ é da forma <b>num</b> ( $n$ ):	$\text{comp}(E) \triangleq < \text{sipush } n >$
se $E$ é da forma <b>add</b> ( $E'$ , $E''$ ):	$s1 = \text{comp}(E'); s2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{iadd} >$
se $E$ é da forma <b>mul</b> ( $E'$ , $E''$ ):	$v1 = \text{comp}(E'); v2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{imul} >$
se $E$ é da forma <b>sub</b> ( $E'$ , $E''$ ):	$v1 = \text{comp}(E'); v2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{isub} >$
se $E$ é da forma <b>div</b> ( $E'$ , $E''$ ):	$v1 = \text{comp}(E'); v2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{idiv} >$

# Compilador de CALCI

- Alocação (no heap) de uma stack frame para guardar os valores dos identificadores declarados.
- A cada momento é conhecido o static link (SL), guardado numa variável local. É possível sempre saber o tipo desse stackframe
- Cada stack frame tem uma estrutura diferente e pré-determinada.

## **decl**

x1 = E1

x2 = E2

...

xn = En

## **in**

E

**.class** *frame\_id*

**.super** java/lang/Object

**.field** public SL *Lancestor\_frame\_id*;

**.field** public x\_0 *type*;

**.field** public x\_1 *type*;

..

**.field** public x\_1 *type*;

**.end** method



# Compilador de CALCI

- Alocação (no heap) de uma stack frame para guardar os valores dos identificadores declarados.
- A cada momento é conhecido o static link (SL), guardado numa variável local. É possível sempre saber o tipo desse stackframe
- Cada stack frame tem uma estrutura diferente e pré-determinada.

## **decl**

x1 = E1

x2 = E2

...

xn = En

## **in**

E

**new** *frame\_id*

**dup**

**invokespecial** *frame\_id*/*<init>()*V

**dup**

**aload** SL

**putfield** *frame\_id*/SL *Lframe\_up*;

**dup**

[[ E1 ]]

**putfield** *frame\_id*/loc\_00 *type*;

**astore** SL

...

[[ E ]]

**aload** SL

**checkcast** *frame\_id*

**getfield** *frame\_id*/SL *Lframe\_up*;

**astore** SL

# Compilador de CALCI

- Alocação (no heap) de uma stack frame para guardar os valores dos identificadores declarados.
- A cada momento é conhecido o static link (SL), guardado numa variável local. É possível sempre saber o tipo desse stackframe
- Cada stack frame tem uma estrutura diferente e pré-determinada.

**decl**

x1 = E1

**in decl**

x2 = E2

**in**

x1 + x2

**aload SL**

**checkcast** *frame\_id*

**getfield** *frame\_2/SL Lframe\_1;*

**getfield** *frame\_1/loc\_00 I*

**aload SL**

**checkcast** *frame\_2*

**getfield** *frame\_2/loc\_00 I*

**iadd**

# Compilador de CALCI

- Em cada contexto, um identificador é localizado por um par: o número de desreferenciações (saltos), e a localização na frame (offset).
- O ambiente guarda e calcula o par de endereços necessário.

**decl**

x1 = E1

**in decl**

x2 = E2

**in**

x1 + x2

**aload SL**

**checkcast** *frame\_id*

**getfield** *frame\_n*/SL *Lframe\_n1*;

**getfield** *frame\_n2*/SL *Lframe\_n2*;

**getfield** *frame\_n2*/SL *Lframe\_n3*;

...

**getfield** *frame\_1*/loc\_X l