Interpretação e Compilação de Linguagens (de Programação)

21/22

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

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Naming

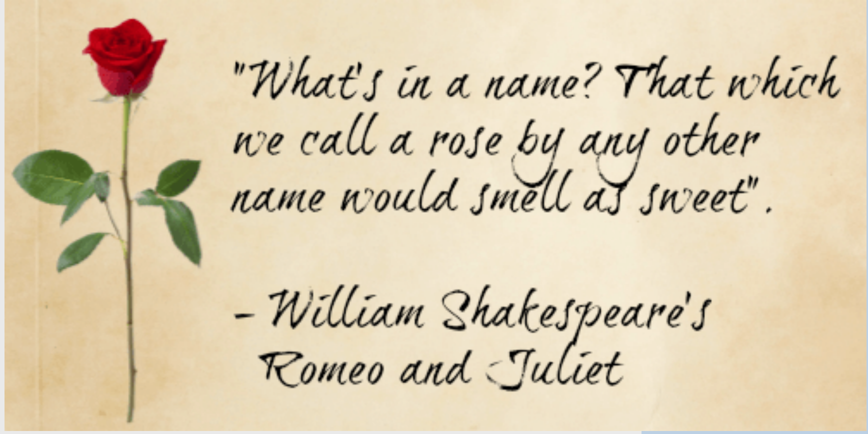
Names are the first tool one uses to introduce abstraction in a programming language (and any language in fact!).

Names allows us to refer to complex things in a concise way!

A name / identifier used in some expression or program always denotes a value previously defined.

Fundamentally, the meaning of a program fragment with names is obtained by replacing each name with the value assigned to it in its definition.

- Literals versus names
- Binding (declaration) of names
- Scope of a definition
- Occurrences of names (free, bound, binding)
- Open and closed code fragments
- Fundamental construct **def** id=E **in** E **end**.
- Language with definitions: CALCI.
- Interpreter using substitution
- Interpreter using environments



Naming Syntax

- Literals
  - Denote fixed values in every context of occurrence
  - Java: `true, false, “foo”, float`
  - OCAML: `true, false, []`
  - C: `1, 1.0, 0xFF, “hello”, int`
- Identifiers
  - Denote values that depend of the context of occurrence
  - In programming languages, identifiers are names for defined constants, variables, functions, methods, classes, modules, types, etc…
  - Java: `x2, y, Count, System.out`
  - C: `printf`

Binding and Scope

- The association between an identifier and the value it denotes is called a *binding*.
- A *binding* between an identifier to the value associated is always established in a well-defined syntactical context (some zone of the program text) and is created by a program construct called a declaration
- The syntactical context (zone of the program text) in which the binding is established is called the *scope* of the binding / declaration.

Binding and Scope

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

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

Binding and Scope

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

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

scope of the binding

Binding and Scope

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

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

Binding and Scope

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

scope of the binding

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

Parts of a Scope

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

Binding and Bound Occurrences

- Occurrences of name **x**

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

Binding occurrences

Binding and Bound Occurrences

- Occurrences of name **x**

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

bound occurrences

Bound Occurrences

- For each bound occurrence there is one and only one binding occurrence (de one occurring in the declaration)

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

Bound Occurrences

- For each bound occurrence there is one and only one binding occurrence (de one occurring in the declaration)

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

Bound Occurrences

- For each bound occurrence there is one and only one binding occurrence (de one occurring in the declaration)

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

Bound Occurrences

- For each bound occurrence there is one and only one binding occurrence (de one occurring in the declaration)

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

Bound Occurrences

- For each bound occurrence there is one and only one binding occurrence (de one occurring in the declaration)

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

Free occurrences

- Any occurrence of an identifier that is not binding nor bound is said **free**

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

Open and Closed fragments

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

```
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
```

Open and Closed fragments

- A program fragment is said to be **open** if it contains free occurrences of identifiers
- Otherwise, a program fragment is said to be **closed** that is, if it does mot contain free occurrences of identifiere
- Open fragments (examples):

free occurrence

C

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

free occurrence

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

OCaml

Semantics of open fragments

- The meaning of a program fragment can only be computed if the value of evert free identifier is known.
- The definition of a compositional semantics for languages with declared identifiers has to consider open fragments.
  For instance, the C block

```
{ int  x  =  2  ;    x  =  x+2  }
```

  is closed but contains open fragments (e.g., x+2).

- In general a complete program (closed fragment) contains open fragments (inside declarations).

Environment

- A closed program necessarily provides bindings all free ocorrentes that inside it, (they must appear in the scope of declarations!).

Given any fragment $\mathcal{E}$ inside a program $\mathcal{P}$, we call **environment of $\mathcal{E}$ in $\mathcal{P}$** to the set of all bindings in which scope $\mathcal{E}$ occurs.

Environment (Quiz)

- What is the environment of subexpression "`x+1`"?

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

Environment (Quiz)

- What is the environment of subexpression "**`x+1`**"?

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

x -> par(0)

Environment (Quiz)

- What is the environment of subexpression "`z+=x`"?

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

Environment (Quiz)

- What is the environment of subexpression "**z+=x**"?

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

x -> par(0)
z -> loc(0)

Environment (Quiz)

- What is the environment of subexpression "`return z`"?

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

Environment (Quiz)

- What is the environment of subexpression "`return z`"?

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

z -> loc(0)

The language CALCI

- CALCI extends our basic expression language CALC with general declarations **def**:

```
def Id = Exp1 in Exp2 end
```

In a **def** expression the first occurrence of *Id* **is** binding, with scope *Exp2*

- A CALCI program is a closed expression of CALCI.

Example:

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

The language CALCI (abstract syntax)

- CALCI AST constructors: num, add, mul, div, sub, id, def

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
def:	String $\times$ CALCI $\times$ CALCI $\rightarrow$ CALCI

The language CALCI (concrete syntax)

- CALCI AST constructors: num, add, mul, div, sub, id, def

```
def x = 2 in
  (def x = x+2
     in
        x + x
   end) + x
end
```

The language CALCI (concrete syntax)

- AST CALCI com os construtores: num, add, mul, div, sub, id, def

```
def x = 2 in
  def y = def z = x+2 in z+z end
  in
     y + def y = 2+x in y end
 end
end
```

Semantics of CALCI (first definition)

The semantics of CALCI may be defined by giving a computable function I which assigns a definite meaning to each program (fragment)

$$\mathrm{I} : \mathrm{CALC} \rightarrow \mathrm{Integer}$$

CALC = set of all programs (closed)

DENOT = set of all meanings (denotations)

CALC Interpreter (evaluation map)

- Algorithm eval(E) that computes the denotation (integer value) of any CALCI expression:

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

eval(**num**(*n*))	≜ *n*
eval(**add**(E1,E2))	≜ **eval**(E1) + **eval**(E2)
eval(**mul**(E1,E2))	≜ **eval**(E1) * **eval**(E2)
…	
eval(**def**(s, E1, E2))	≜ { V = eval(E1); G = **substv**(s, E2,V); **eval**(G); }

Fundamentally, the meaning of a program with names is always obtained by replacing each name with the value assigned to it in its definition.

The substitution function

substv(s, E, V)

Computes the expression (AST) that results from replacing in program (AST) E all free occurrences of identifier s by value V.

Examples (what does substv do?)

subst(s, s+s+2, 4) = 4+4+2

subst(y, def x=y in def y=2 in x+y, 2) =
def x=2 in def y=2 in x+y

Definition of Substv function (on ASTs)

substv(s, **num**(*n*), V) ≜ num(*n*);

substv(s, **id**(*s*), V) ≜ V;

substv(s, **add**(E1, E2),F) ≜ add(**substv**(s, E1, V), **substv**(s, E2 , V));

…

substv(s, **def**(s', E1, E2), V) ≜ **if** s = s'
{ G = **substv**(s, E1, V);
def(s, G, E2); }
else
{ G = **substv**(s, E1, V);
def(s', G, **substv**(s, E2, V)); }

CALC Interpreter (evaluation map)

- Algorithm eval(E) that computes the denotation (integer value) of any CALCI expression:

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

eval(**num**(*n*))	≜ *n*
eval(**add**(E1,E2))	≜ **eval**(E1) + **eval**(E2)
eval(**mul**(E1,E2))	≜ **eval**(E1) * **eval**(E2)
…	
eval(**def**(s, E1, E2))	≜ { V = eval(E1); G = **substv**(s, E2,V); **eval**(G); }

- Note: we don't need to define the case **eval**(**id**(s)). Why ?

Semantics of CALCI (better definition)

- The substitution-based semantics of CALCI is very simple and intuitive from the perspective of specification because it is very simple, and conforms to the essential meaning of names.

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

- However, it is not efficient, requires runtime manipulation of ASTs and does not scale well for compilation.
- Using a notion of runtime environment (or spaggetti stack) the effect of explicit syntactical substitution can be performed in a lazy way.

Semantics of CALCI (better definition)

- Algorithm eval( ) that computes the denotation (integer value) of any **open** CALCI expression:

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

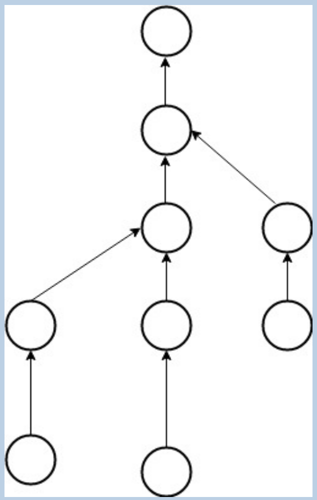
CALCI = open programs

ENV = environments

Integer = meanings (denotations)

The Environment as an ADT

- In practice, it is convenient to implement environments using a mutable stack-like data structure called a "**spaghetti stack**".
- **NOTE**: In block structures languages (eg., in all "decent" modern languages) the addition and remotion of biddings between identifiers and values follows a strict stack LIFO discipline.
- An environment stores all bindings relative to the current scope and all involving scopes in **frames**.
- From any environment state one may create a new "child" frame, corresponding to a new nested scope.
- Each frame links to the ancestor frame using a reference.



The Environment as an ADT

- Environment operations:

```
Environ BeginScope()
```

  - Pushes into the environment a new frame, where new bindings will be stored.
  - A given identifier can only be bound once in a given frame, but may be bound in different frames (to possibly different values).

```
Environ EndScope()
```

  - returns the father environment (pops off top frame).

```
void assoc(String id, Value val)
```

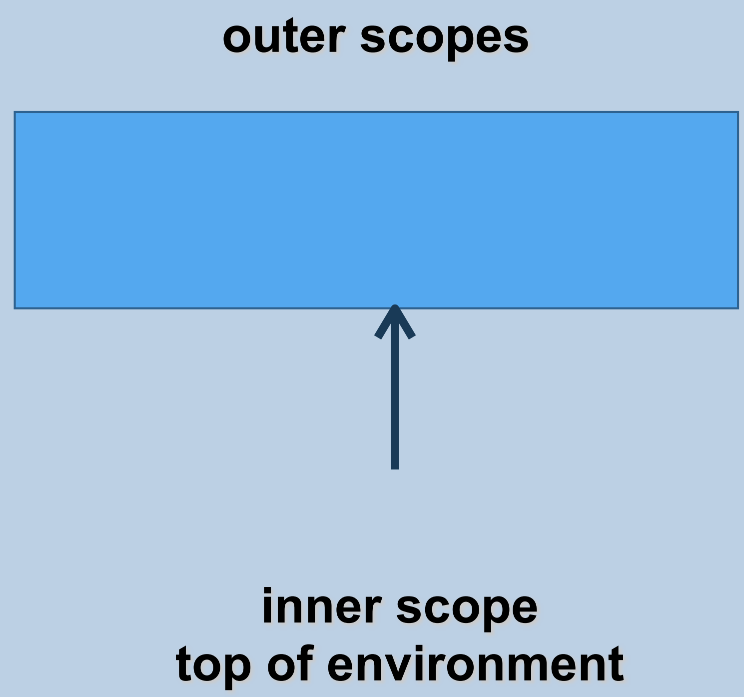
  - Adds a new binding for identifier id to the value val in the top frame of the environment (if id is not bound there yet).

```
Value Find(String id)
```

  - Returns the value associated to id in the environment, as defined by the innermost binding (the binding in the topmost frame that binds id).
  - In practice, **Find** searches for id from top to bottom following the stack frame chain, from "most recent" up, so that the appropriate scoping is respected.

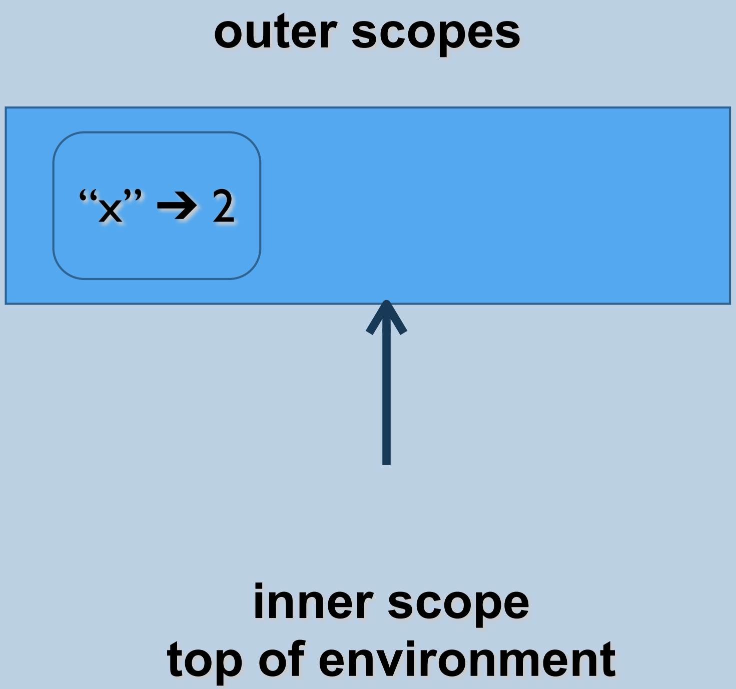
Environment in action

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



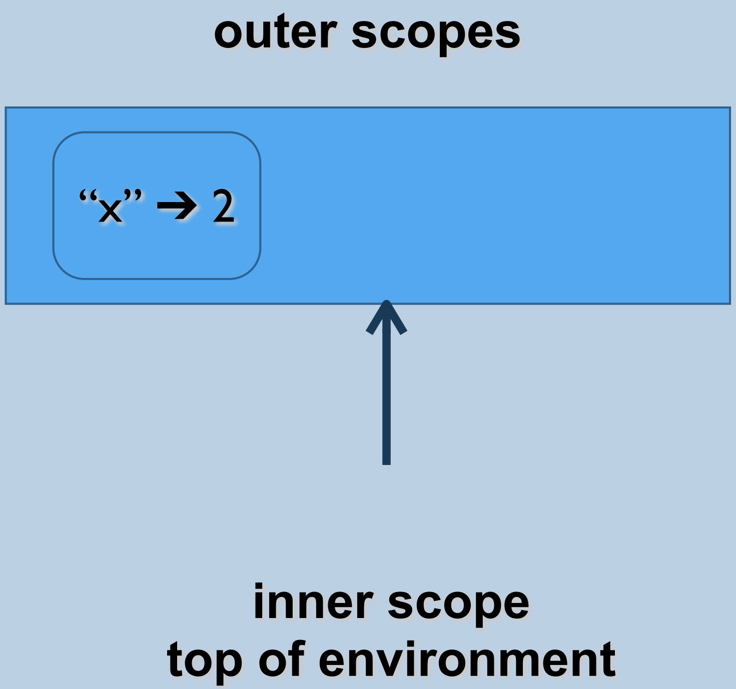
Environment in action

```
env = new Environment();

env.Assoc("x", 2);
```



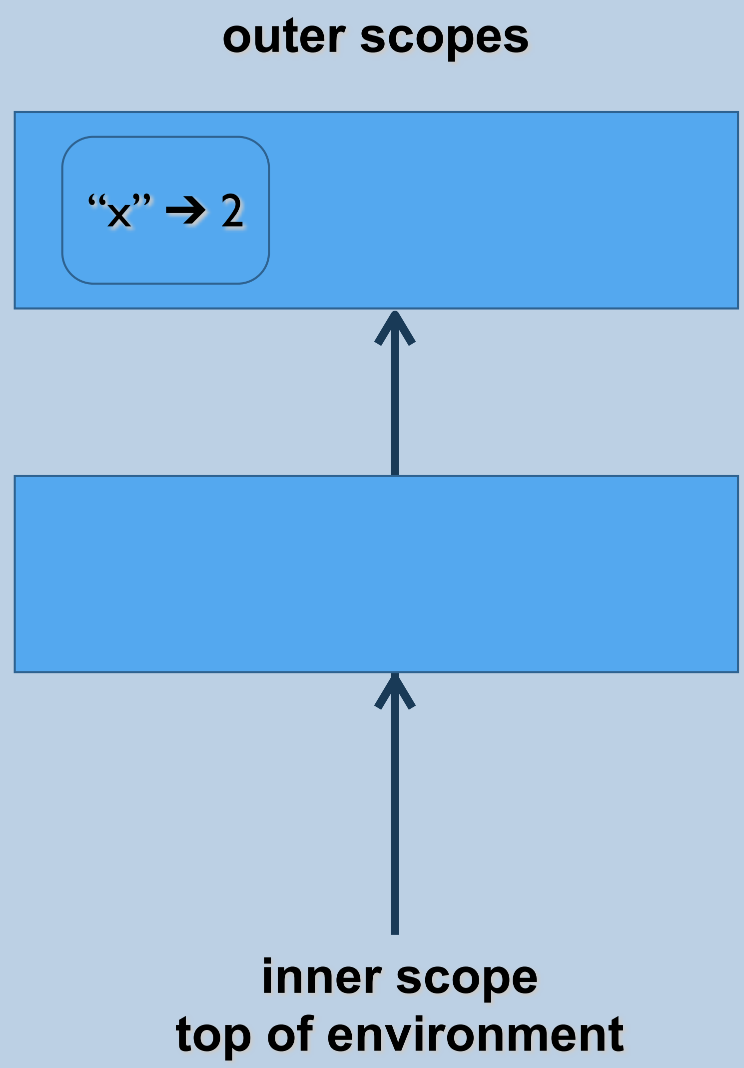
Environment in action

```
env = new Environment();
env.Assoc(“x”, 2);
val = env.Find(“x”);          // returns 2
val = env.Find(“y”);          // raises “Not declared”
```



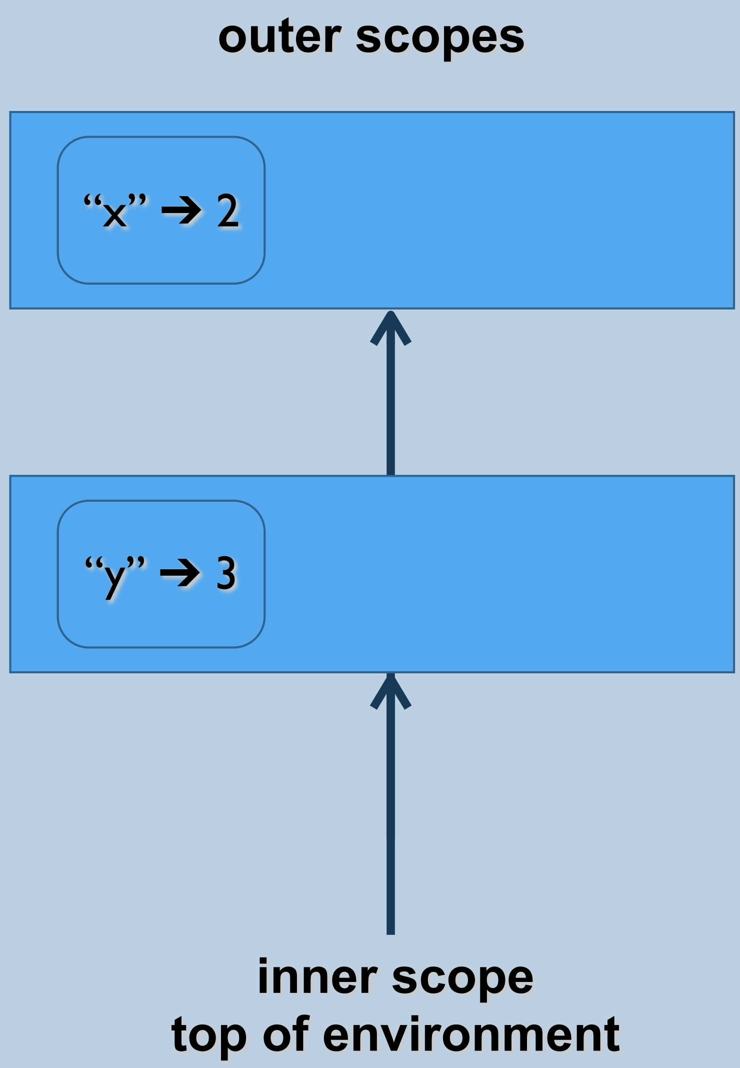
Environment in action

```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x");          // returns 2
val = env.Find("y");          // raises "Not declared"
env = env.BeginScope();
```



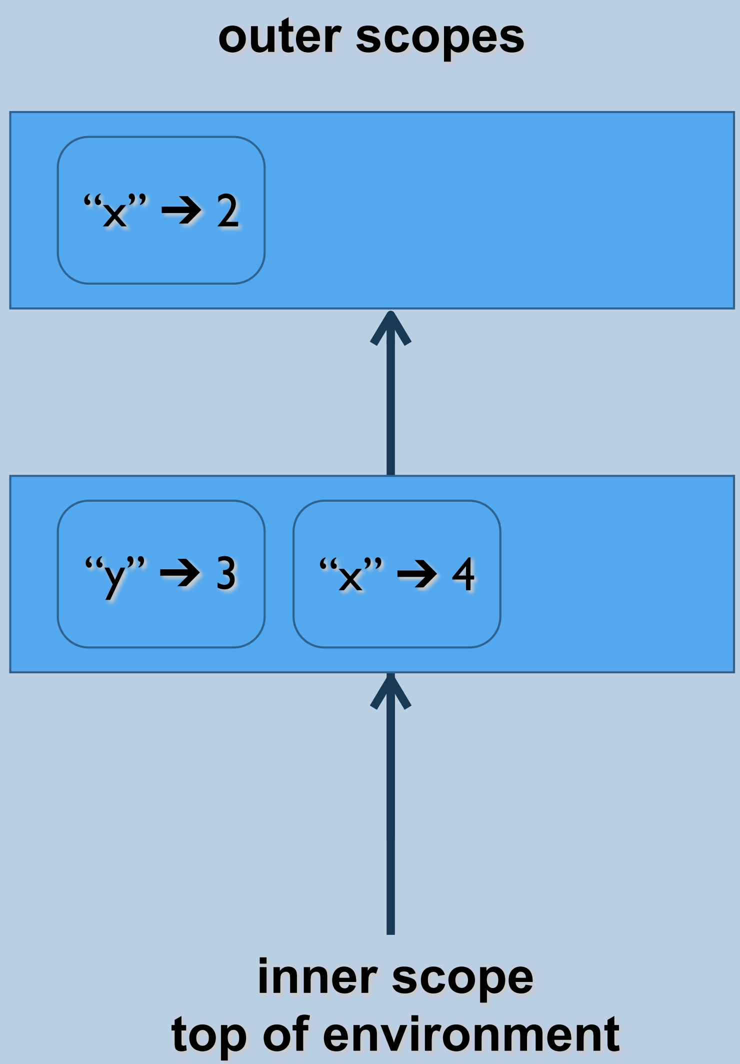
Environment in action

```
env = new Environment();

env.Assoc(“x”, 2);

val = env.Find(“x”);          // returns 2

val = env.Find(“y”);          // raises “Not declared”

env = env.BeginScope();

env.Assoc(“y”, 3);
```



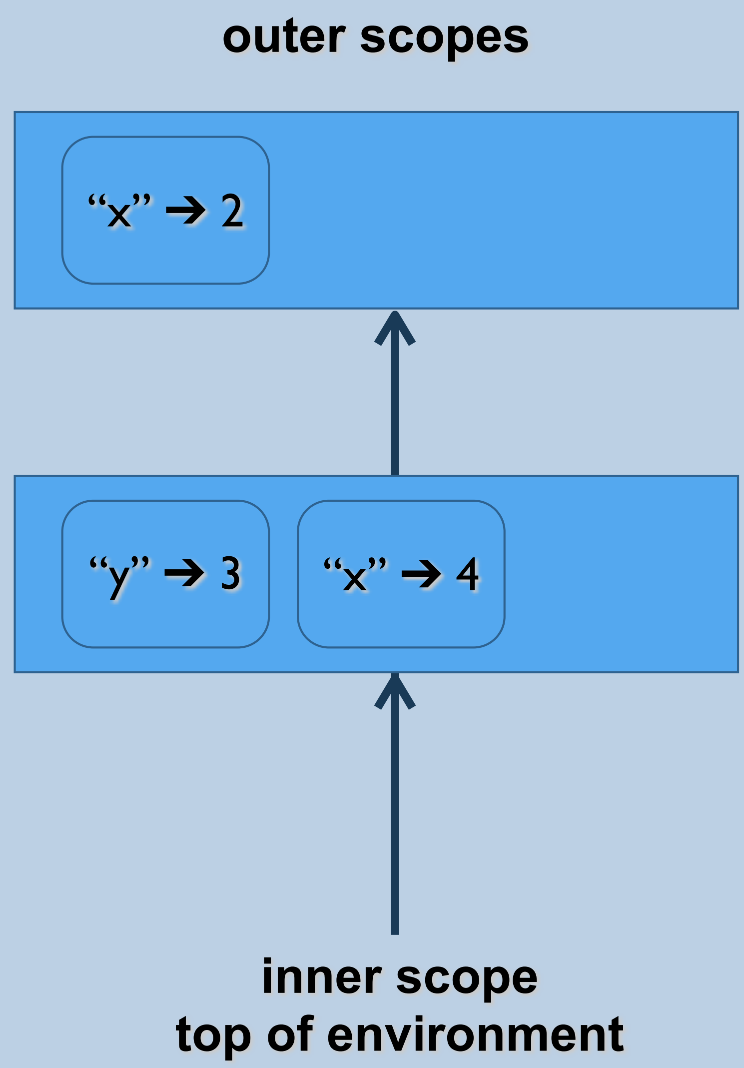
Environment in action

```
env = new Environment();
env.Assoc(“x”, 2);
val = env.Find(“x”);          // returns 2
val = env.Find(“y”);          // raises “Not declared”
env = env.BeginScope();
env.Assoc(“y”, 3);
env.Assoc(“x”, 4);
```



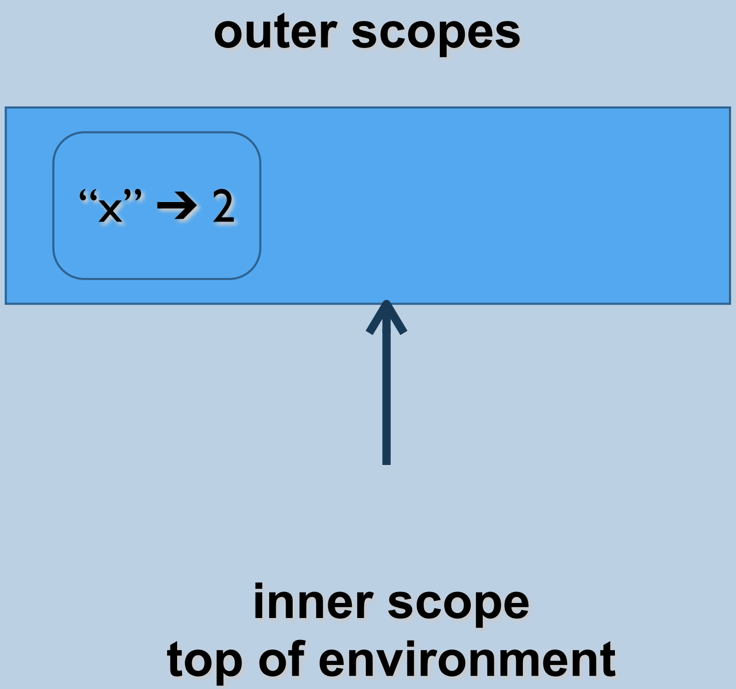
Environment in action

```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x");          // returns 2
val = env.Find("y");          // raises "Not declared"
env = env.BeginScope();
env.Assoc("y", 3);
env.Assoc("x", 4);
val = env.Find("y");          // returns 3
val = env.Find("x");          // returns 4
```



Environment in action

```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x");         // returns 2
val = env.Find("y");         // raises "Not declared"
env = env.BeginScope();
env.Assoc("y", 3);
env.Assoc("x", 4);
env.Assoc("y", 0);         // raises "Declared twice
val = env.Find("y");         // returne 3
val = env.Find("x");         // returns 4
env=env.EndScope()
val = env.Find("x")         // returns 2
```



CALC Interpreter (environment based)

- Algorithm eval( ) that computes the denotation (integer value) of any **open** CALCI expression:

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

eval(**num**(*n*) , *env*) ≜ *n*

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

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

…

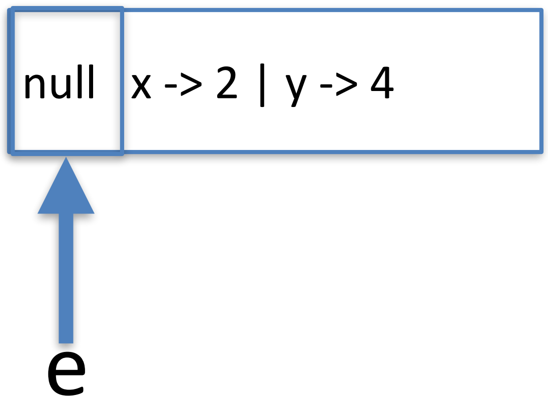
eval(**def**(s, E1, E2), *env*) ≜ **[** v1 = **eval**(E1, *env*);
env = *env*.BeginScope();
env = *env*.Assoc(s, v1);
val = **eval**(E2, *env*);
env = *env*.EndScope();
return val **]**

- Note: Case of id(s) implemented by lookup of the value of s in the current environment

Sample execution (environment actions)

```
def x = 2
    y = x+2 in
def z = 3 in
 def y = x+1 in
  x + y + z end end end;;
```

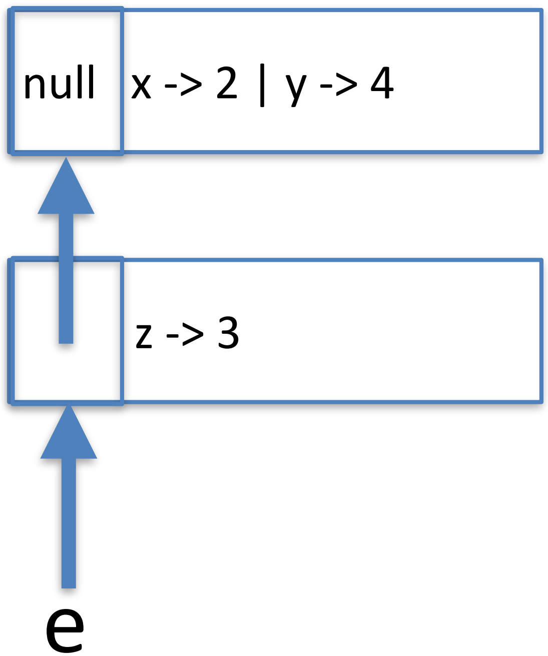
Sample execution (environment actions)




```
def x = 2
    y = x+2 in
def z = 3 in
 def y = x+1 in
  x + y + z end end end;;
```

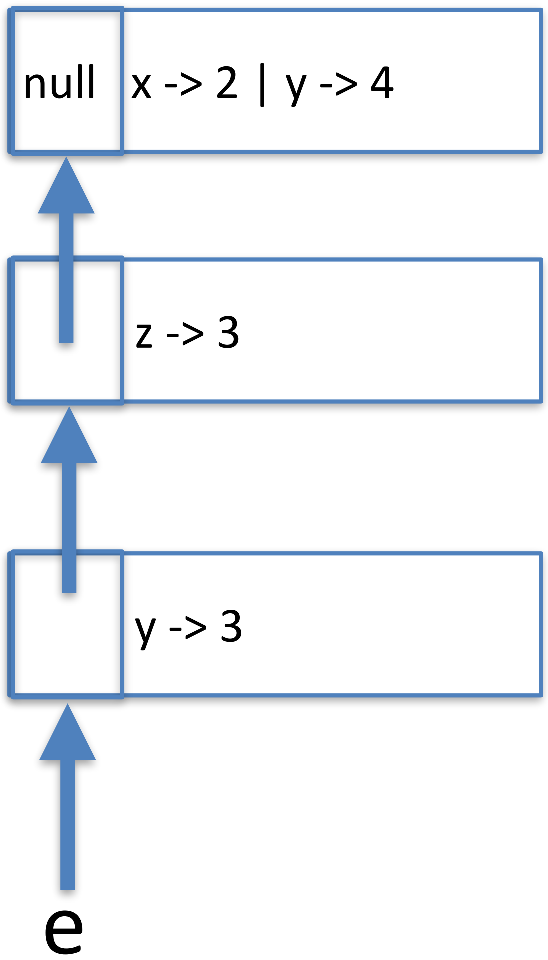
Sample execution (environment actions)

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```



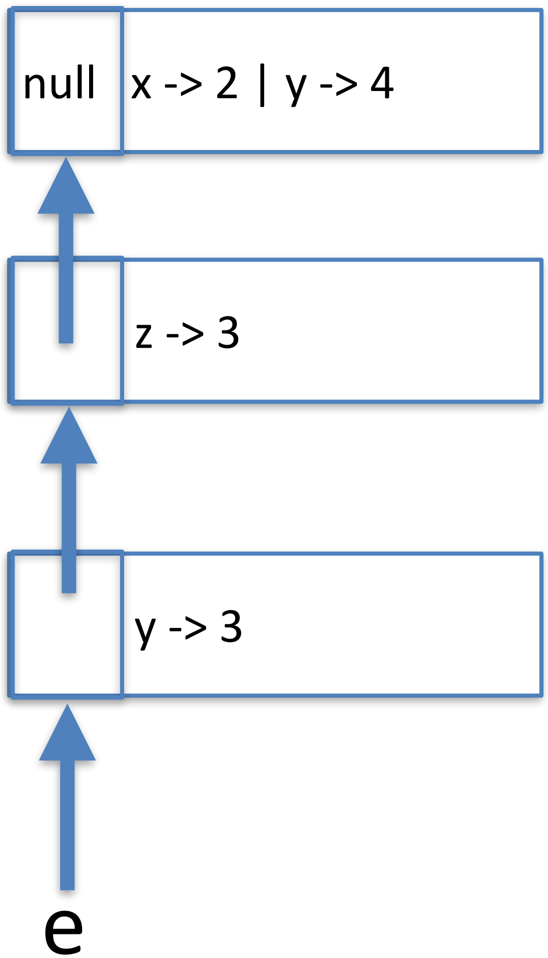
Sample execution (environment actions)

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```



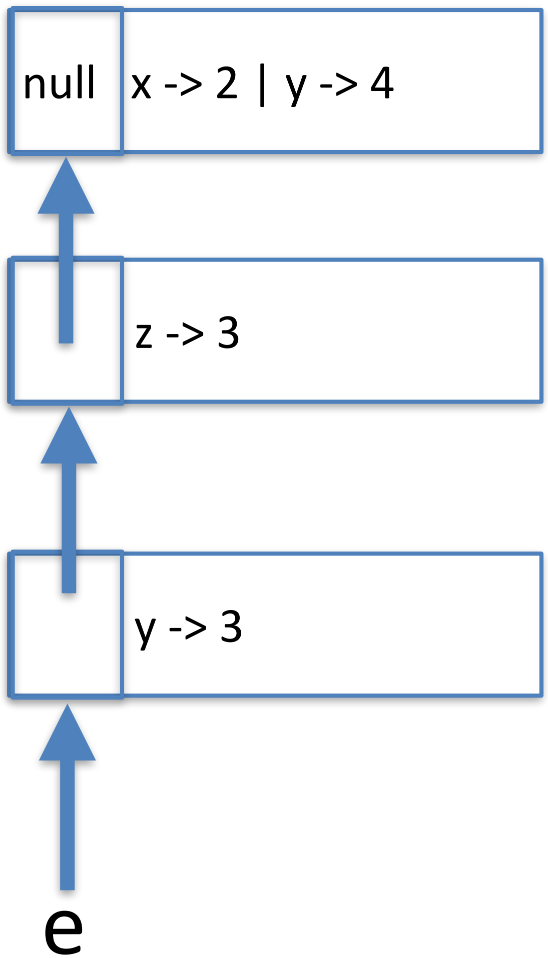
Sample execution (environment actions)

```
def x = 2
    y = x+2 in
def z = 3 in
 def y = x+1 in
  x + y + z end end end;;
```



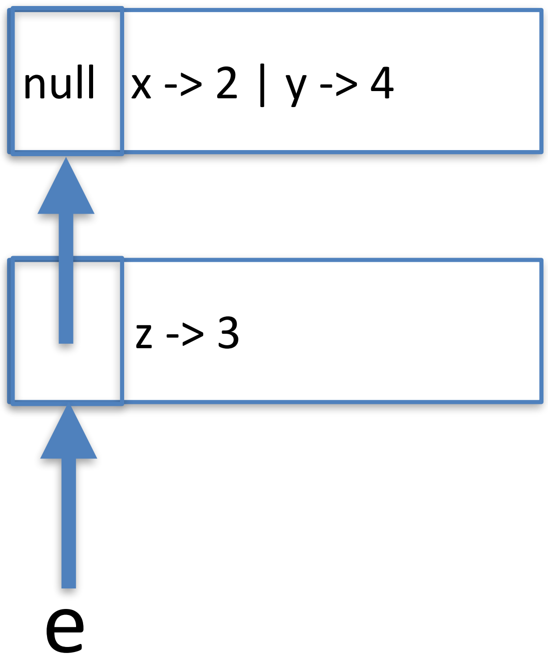
Sample execution (environment actions)

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
   x + y + z end end end;;
```

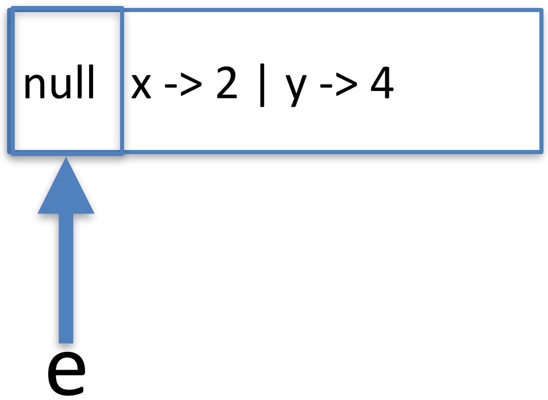


Sample execution (environment actions)

```
def x = 2
     y = x+2 in
def z = 3 in
 def y = x+1 in
  x + y + z end end end;;
```



Sample execution (environment actions)




```
def x = 2
    y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```

Sample execution (environment actions)

```
def x = 2
    y = x+2 in
def z = 3 in
 def y = x+1 in
  x + y + z end end end;;
```

Interpretação e Compilação de Linguagens (de Programação)

21/22

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

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa