

Implementazione dell'interprete di un nucleo di linguaggio funzionale

Concetti alla base della definizione di un ambiente per l'interprete

Entità denotabili

- **Entità denotabili:** elementi di un linguaggio di programmazione a cui si può **assegnare un nome**:
 - Entità i cui nomi sono **definiti dal linguaggio** di programmazione (tipi primitivi, operazioni primitive, ...)
 - Entità i cui nomi sono **definiti dall'utente** (variabili, parametri, procedure, tipi, costanti simboliche, classi, oggetti, ...)

Binding e scope

- Un **binding** è un **legame**, ovvero un'associazione tra un nome e una entità del linguaggio (funzione, struttura dati, oggetto, etc.)
- Lo **scope** (o **ambito di visibilità**) di un binding definisce quella parte del programma nella quale il binding è attivo

Binding statico & dinamico

- **static binding**: i legami nome-entità sono definiti *prima* dell'esecuzione del programma
- **dynamic binding**: i legami nome-entità sono definiti *durante* l'esecuzione del programma

Ambiente

L'**ambiente** è definito come

- **l'insieme delle associazioni nome-entità** esistenti a run-time in uno specifico punto del programma e in uno specifico momento dell'esecuzione
- *Nella macchina astratta del linguaggio, per ogni nome e per ogni sezione del programma, l'ambiente determina l'associazione corretta*

Ambiente e dichiarazioni

- Le **dichiarazioni** sono il costrutto linguistico che permette di introdurre associazioni nell'ambiente

```
int x;  
int f( ) {  
    return 0;  
}
```

Dichiarazione di una
variabile

Dichiarazione di una
funzione

Dichiarazione di
tipo

```
type BoolExp =  
    | True  
    | False  
    | Not of BoolExp  
    | And of BoolExp*BoolExp
```

Ambiente e dichiarazioni

```
{ int x;  
  x = 22;  
  { char x;  
    x = 'a';  
  }  
}
```

Qui lo stesso nome, la variabile x, denota due entità differenti

Valore intero

Carattere

Ambiente e dichiarazioni

```
Class A { ... }
```

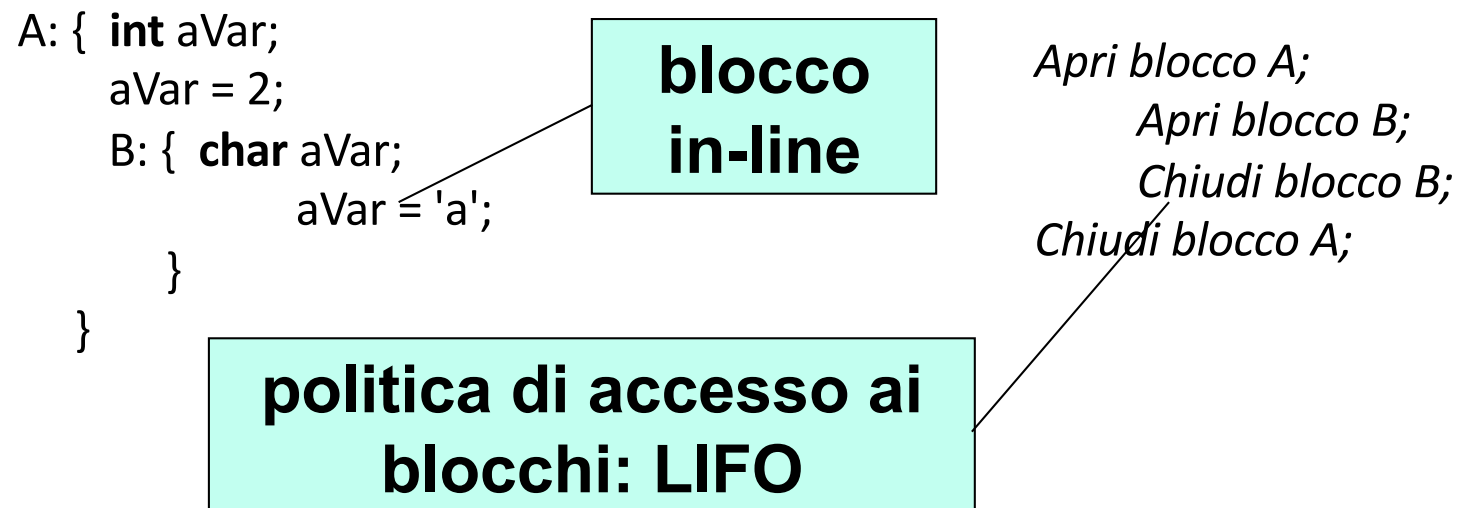
```
A a1 = new A( );  
A a2 = a1;
```

Aliasing: nomi diversi
per lo stesso oggetto
(tramite riferimenti)

Blocchi

- Un **blocco** è una **regione testuale** del programma che può contenere dichiarazioni
 - **C, Java:** { ... }
 - **OCaml:** let ... in
- Un blocco può essere:
 - **associato a una funzione:** corpo della funzione con le dichiarazioni dei parametri formali
 - **in-line:** meccanismo per raggruppare comandi (ad es. corpo di un ciclo)

Blocchi



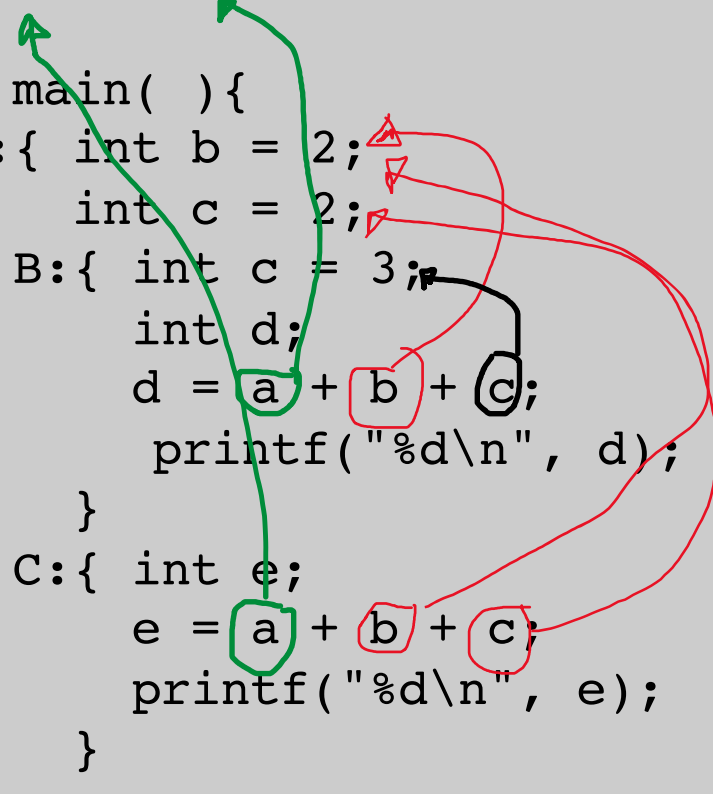
*I cambiamenti dell'ambiente avvengono principalmente sia all'entrata che all'uscita dai **blocchi** (anche annidati)*

Tipi di ambiente

- ***Ambiente locale***: associazioni dichiarate **localmente** al blocco, comprese le eventuali associazioni relative ai parametri
- ***Ambiente non locale***: associazioni dei nomi che sono **visibili** all'interno del blocco, ma **non dichiarati nel blocco stesso**
- ***Ambiente globale***: associazioni per i nomi **visibili a/ usabili da tutte** le componenti che costituiscono il programma

Tipi di ambiente: esempio in C

```
int a = 1 ;  
  
int main( ) {  
    A: { int b = 2 ;  
        int c = 2 ;  
        B: { int c = 3 ;  
            int d ;  
            d = a + b + c ;  
            printf( "%d\n", d ) ;  
        }  
        C: { int e ;  
            e = a + b + c ;  
            printf( "%d\n", e ) ;  
        }  
    }  
}
```



Ambiente locale del blocco B

- associazioni per c e d

Ambiente non locale per B

- associazione per b, ereditata da A
- associazione globale per a

Ambiente Globale

- associazione per a

Tipi di ambiente: esempio in Java

```
public class Prova {  
    public static void main(String[ ] args) {  
        int a =1 ;  
        A:{ int b = 2;  
            int c = 2;  
            B:{ int c = 3;  
                int d;  
                d = a + b + c;  
                System.out.println(d);  
            }  
            C:{ int e;  
                e = a + b + c;  
                System.out.println(e);  
            }  
        } } }a
```

Tipi di ambiente: esempio in Java

```
public class Prova {  
    public static void main(String[ ] args) {  
        int a =1 ;  
        A:{ int b = 2;  
           int c = 2;  
        B:{ int c = 3;  
            int d;  
            d = a + b + c;  
            System.out.println(d);  
        }  
        C:{ int e;  
            e = a + b + c;  
        }  
    }  
}
```

NB: in **Java** non è possibile ri-dichiarare una variabile già dichiarata in un blocco più esterno

```
$ javac Prova.java
```

```
Prova.java:7: c is already defined in main(java.lang.String[])  
        B:{ int c = 3;  
            ^
```

Scope

- Lo **scope** (o *ambito di visibilità*) di un binding definisce quella parte del programma nella quale il binding è attivo
- Lo scope può essere:
 - **statico** o **lessicale**: cioè determinato dalla struttura sintattica del programma
 - **dinamico**: cioè determinato dalla struttura a tempo di esecuzione
- I due differiscono solo per l'**ambiente non locale**

Scoping statico

Il binding attivo per un nome non locale è determinato dai blocchi che **testualmente** racchiudono a partire da quelli più interni:

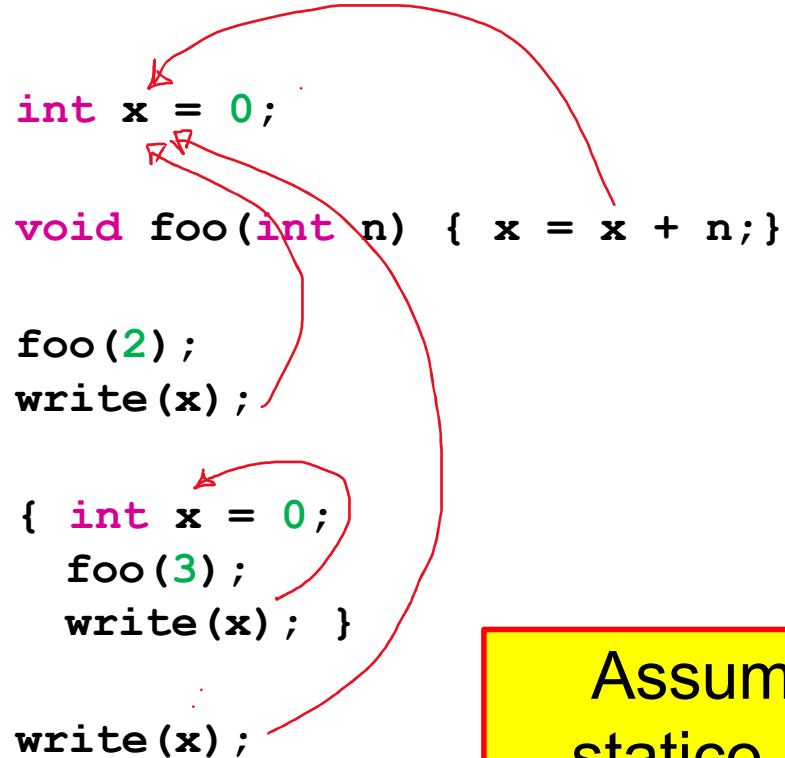
```
int x = 0;

void foo(int n) { x = x + n; }

foo(2);
write(x);

{ int x = 0;
  foo(3);
  write(x); }

write(x);
```



Assumendo scoping
statico, questo codice
stampa... ???

Scoping statico

Il binding attivo per un nome non locale è determinato dai blocchi che **testualmente** racchiudono a partire da quelli più interni:

```
int x = 0;

void foo(int n) { x = x + n; }

foo(2);
write(x);

{ int x = 0;
  foo(3);
  write(x); }

write(x);
```

write(x)

1. x è quella della prima dichiarazione, che `foo(2)` fa passare da 0 a 2
2. x è quella della seconda dichiarazione, che `foo(3)` non tocca perché agisce sulla x della prima dichiarazione
3. x è quella della prima dichiarazione, che `foo(3)` ha fatto passare da 2 a 5

Assumendo scoping statico, questo codice stampa... **2 0 5**

Scoping dinamico

Il binding attivo per un nome non locale è determinato dalla sequenza dei blocchi attivi nello stack, a partire dai blocchi attivati più **recentemente**:

```
int x = 0;
```

```
void foo(int n) { x = x + n; }
```

```
foo(2);  
write(x);
```

```
{ int x = 0;  
  foo(3);  
  write(x); }
```

```
write(x);
```

Assumendo scoping
dinamico, questo codice
stampa... ???

Scoping dinamico

Il binding attivo per un nome non locale è determinato dalla sequenza dei blocchi attivi nello stack, a partire dai blocchi attivati più **recentemente**:

```
int x = 0;
```

```
void foo(int n) { x = x + n; }
```

```
foo(2);
```

```
write(x);
```

```
{ int x = 0;  
  foo(3);  
  write(x); }
```

```
write(x);
```

write(x)

1. x è quella della prima dichiarazione, che foo(2) fa passare da 0 a 2
2. x è quella della seconda dichiarazione, che foo(3) influenza, facendone passare il valore da 0 a 3
3. x è quella della prima dichiarazione, che foo(2) aveva aggiornato a 2 (e foo(3) non aveva toccato)

Assumendo scoping
dinamico, questo codice
stampa... **232**

Cambiamenti dell'ambiente

- L'ambiente può cambiare a **run time**, ma i cambiamenti avvengono di norma in precisi momenti:
 - **entrando in un blocco**
 - creazione delle associazioni fra i nomi locali al blocco e gli oggetti denotati
 - disattivazione delle associazioni per i nomi ridefiniti
 - **uscendo dal blocco**
 - distruzione delle associazioni fra i nomi locali al blocco e gli oggetti denotati
 - riattivazione delle associazioni per i nomi che erano stati ridefiniti
- **N.B.:** Il tempo di vita degli oggetti denotati non è necessariamente uguale al tempo di vita di un'associazione
 - (ad es. con riferimenti/puntatori a memoria dinamica, in caso di aliasing)

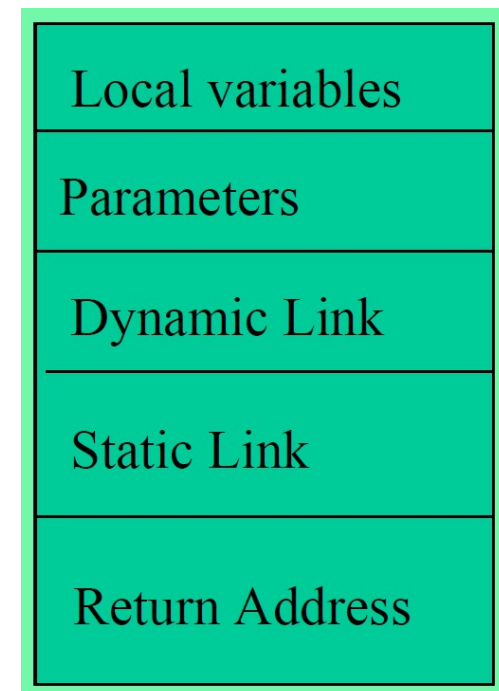
Realizzazione dell'ambiente come stack nel run-time support

Lo stack

Lo **stack** in un linguaggio di programmazione è una struttura dati utilizzata dal run-time support per implementare le funzionalità legate all'ambiente:

Lo **stack** è una **pila di record di attivazione** ognuno dei quali solitamente (per i blocchi associati a funzioni) contiene:

- variabili locali del blocco (incluso il risultato della funzione)
- parametri della funzione
- link al record di attivazione del chiamante
- link al record di attivazione del blocco più esterno (sintatticamente)
- punto nel codice del chiamante in cui tornare al termine della chiamata



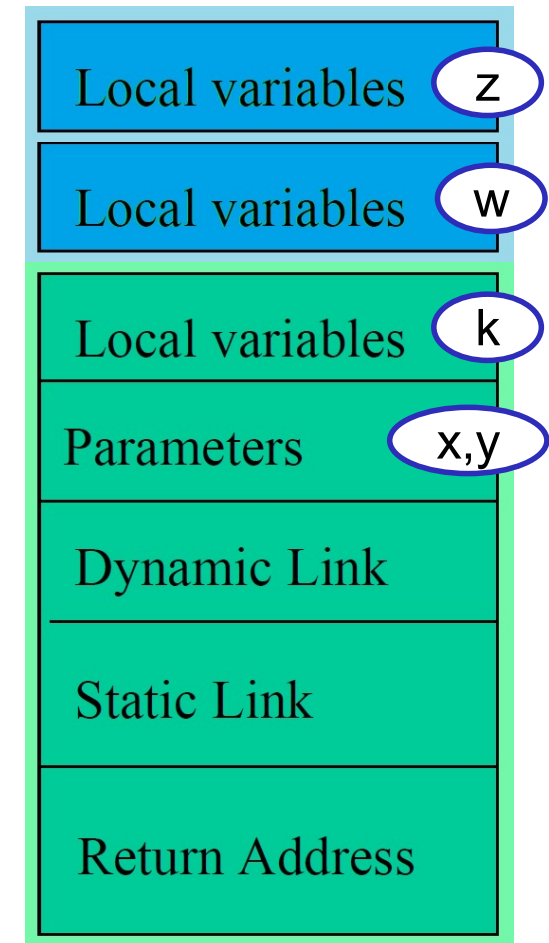
Lo stack

Per i **blocchi in-line, non associati a funzioni**

- la struttura è più semplice (solo variabili locali)
- si impilano sul record di attivazione della funzione che li contiene

Per il momento li ignoriamo

```
void f(int x; int y) {  
    int k;  
    ...  
    { ... int w; ...  
        { ... int z; ... }  
    ...  
}  
}
```



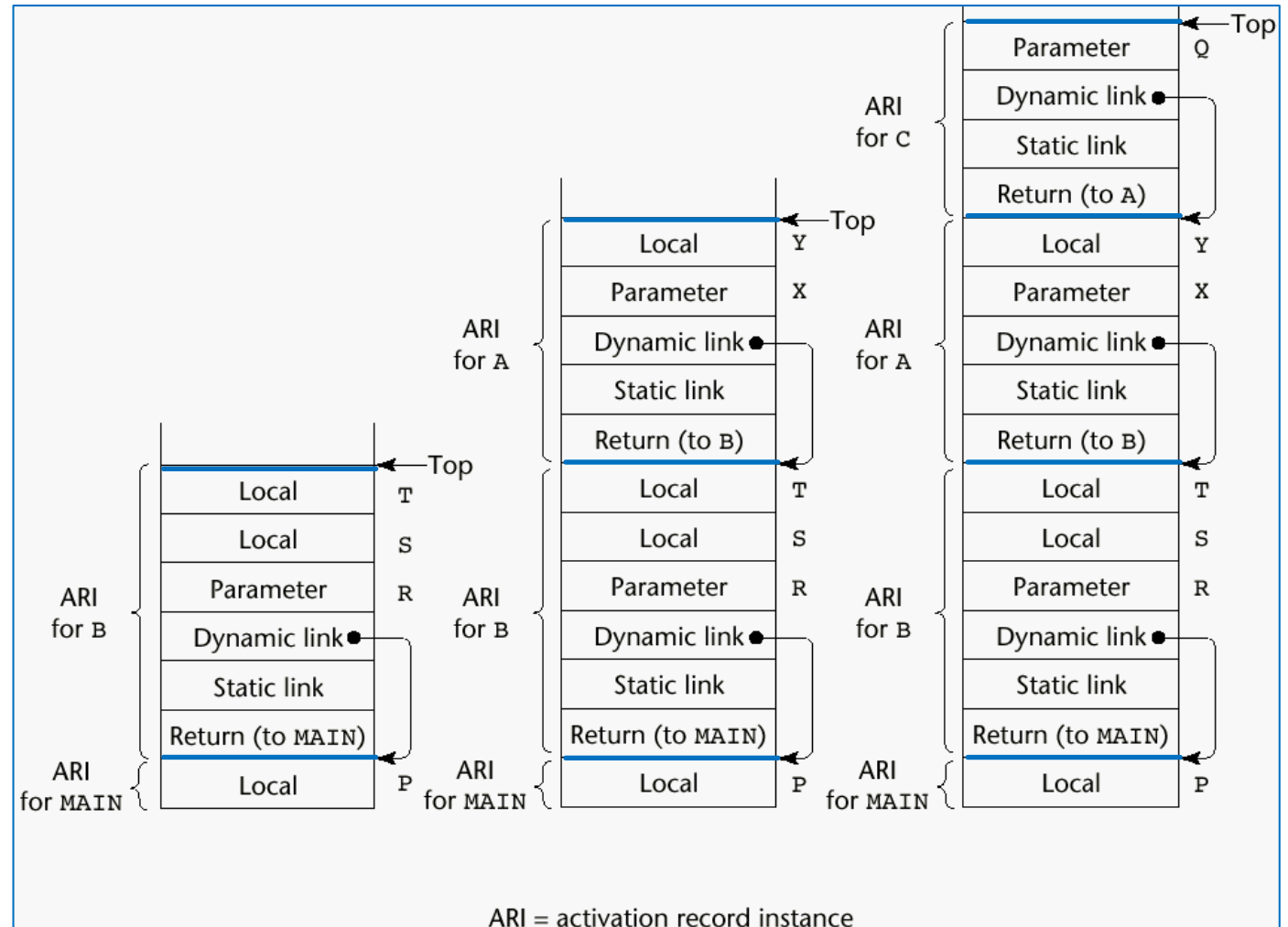
Esempio di esecuzione in C

```
void A(int x) {
    int y;
    C(y);
}

void B(float r) {
    int s, t;
    A(s);
}

void C(int q) {
    ...
}

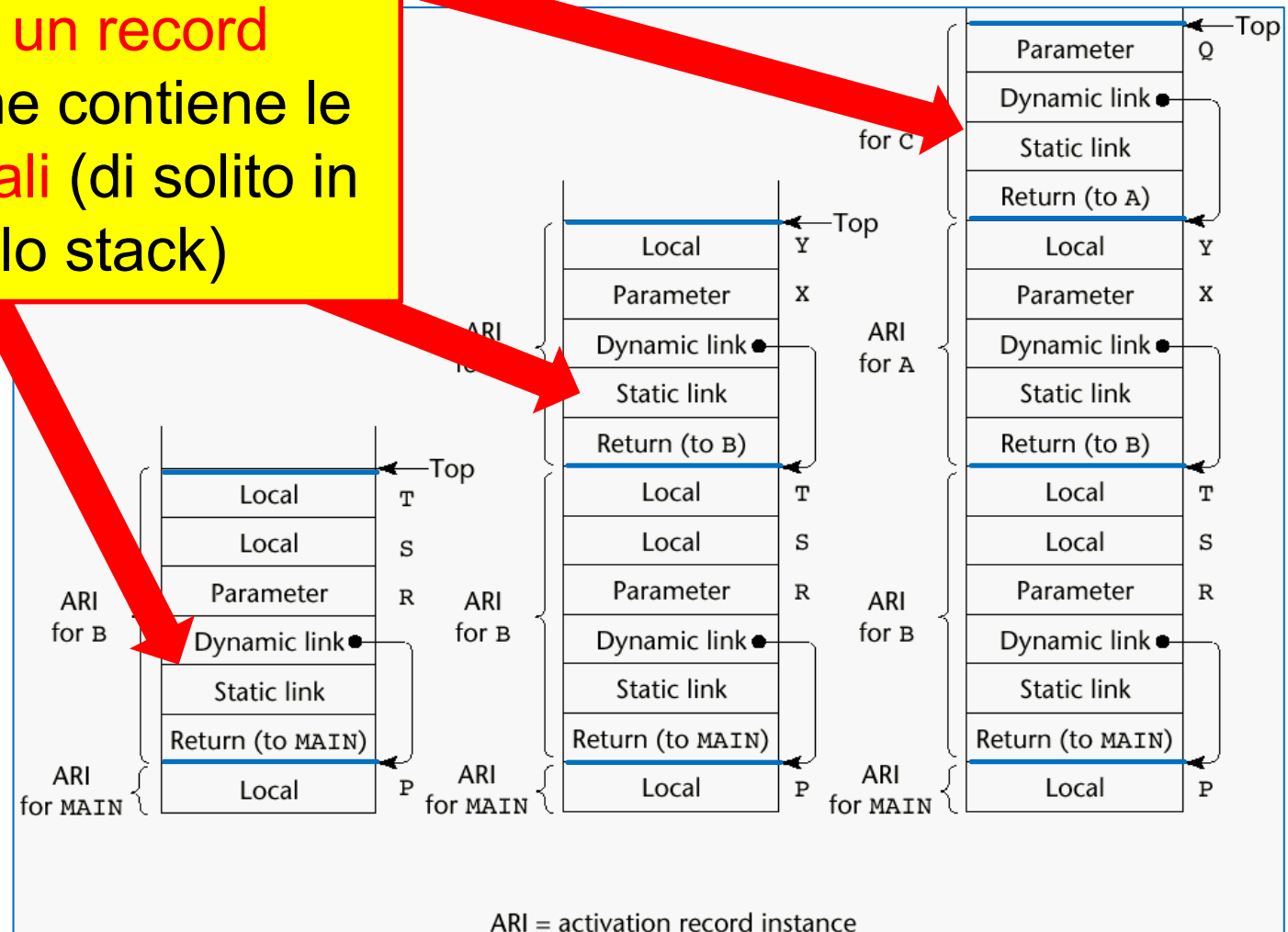
void main() {
    float p;
    B(p);
}
```



In C non si possono definire funzioni annidate, quindi lo **Static Link** punta sempre a un record "speciale" che contiene le variabili globali (di solito in fondo allo stack)

Attivazione in C

```
void
in
C(
}
void
int s, t;
A(s);
}
void C(int q) {
...
}
void main() {
float p;
B(p);
}
```



Il **Dynamic Link** serve a ripristinare il puntatore **Top** al termine dell'esecuzione della funzione, quando il record attivo viene rimosso

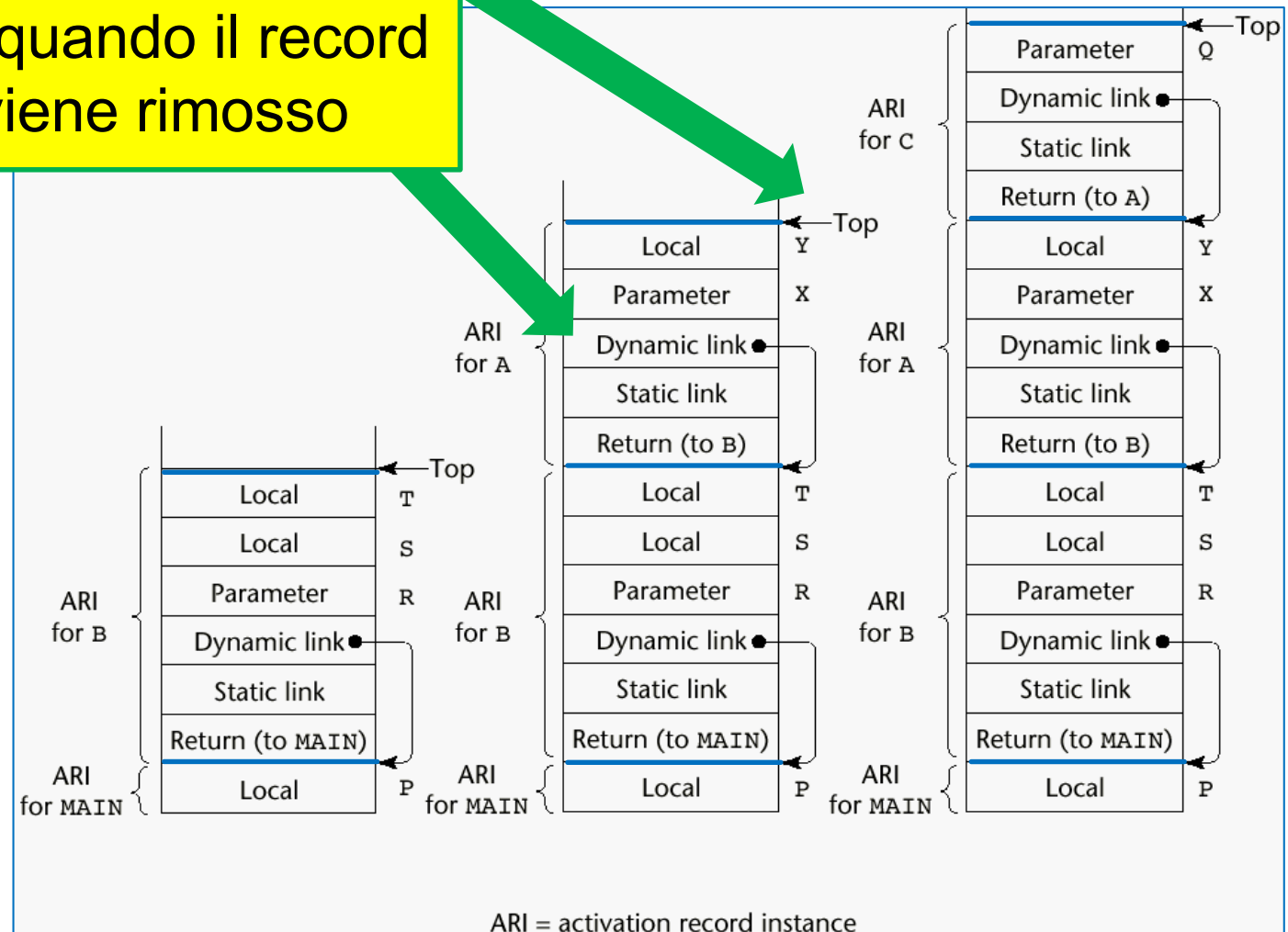
ne in C

```
void A(
    int y
) {
    C(y);
}

void B(float r) {
    int s, t;
    A(s);
}

void C(int q) {
    ...
}

void main() {
    float p;
    B(p);
}
```

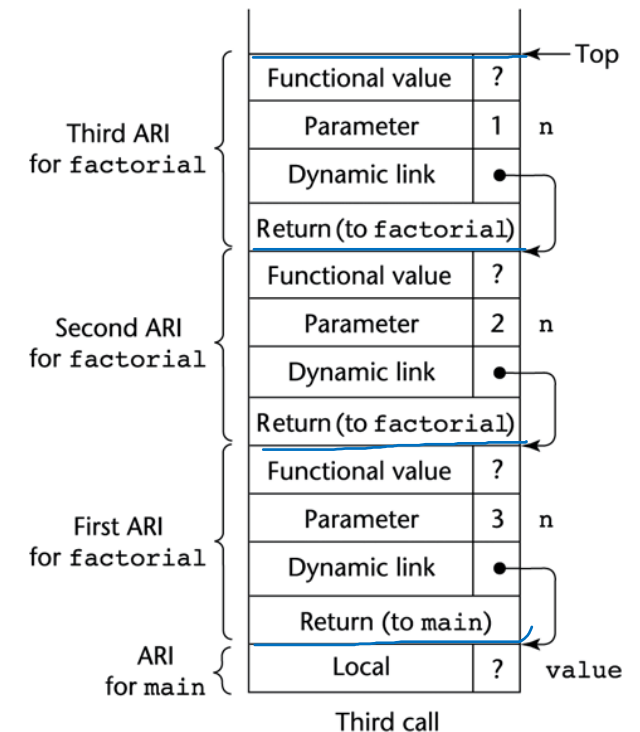
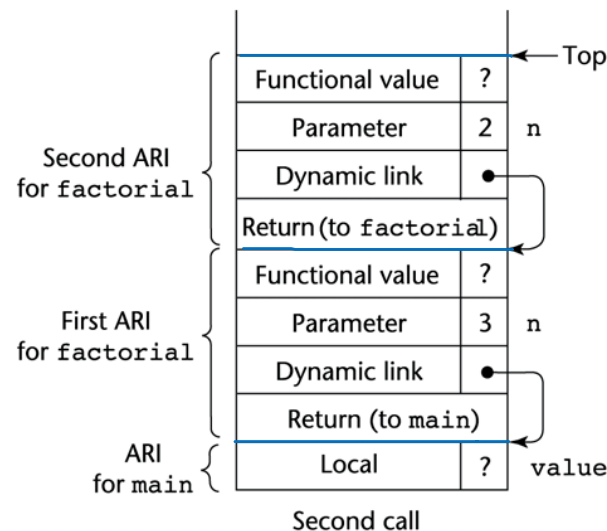
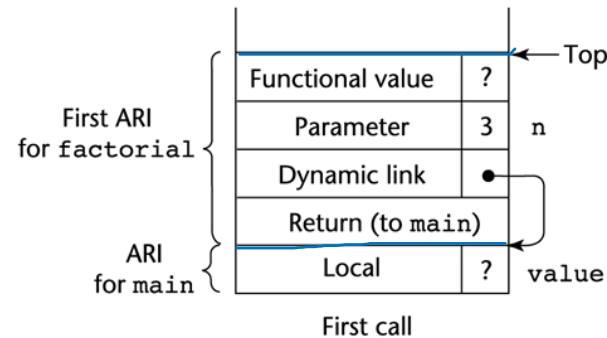


Esempio ricorsivo in C

```
int factorial(int n)
{
    if(n<=1) return 1;
    else return
        n*factorial(n-1);
}

void main( )
{
    int value;
    value = factorial(3);
}
```

"Functional value" è
il risultato della
funzione

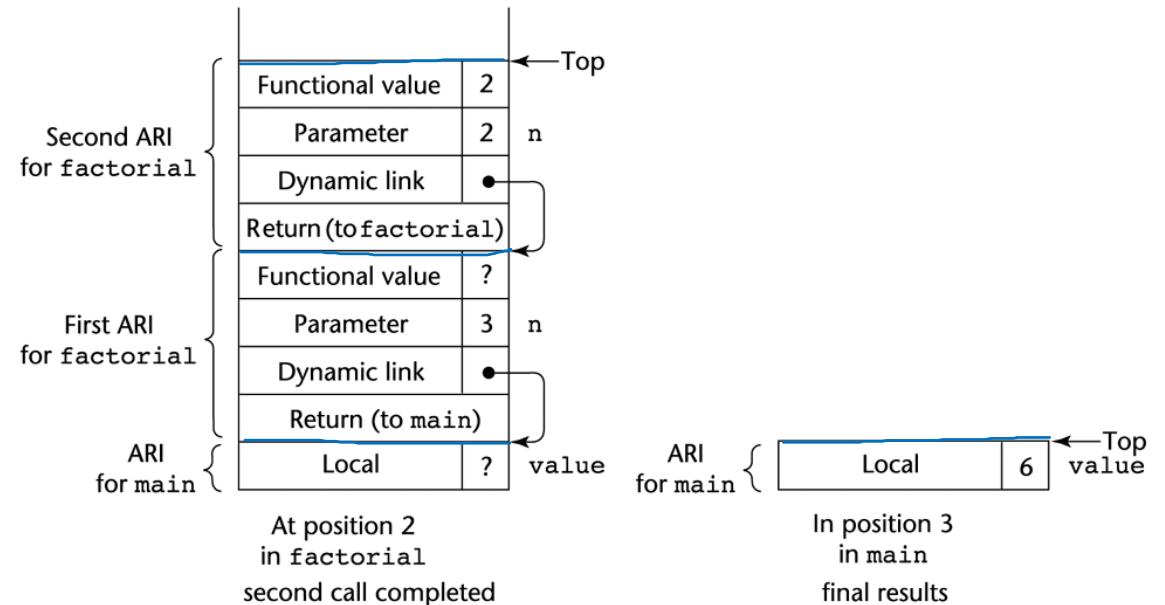
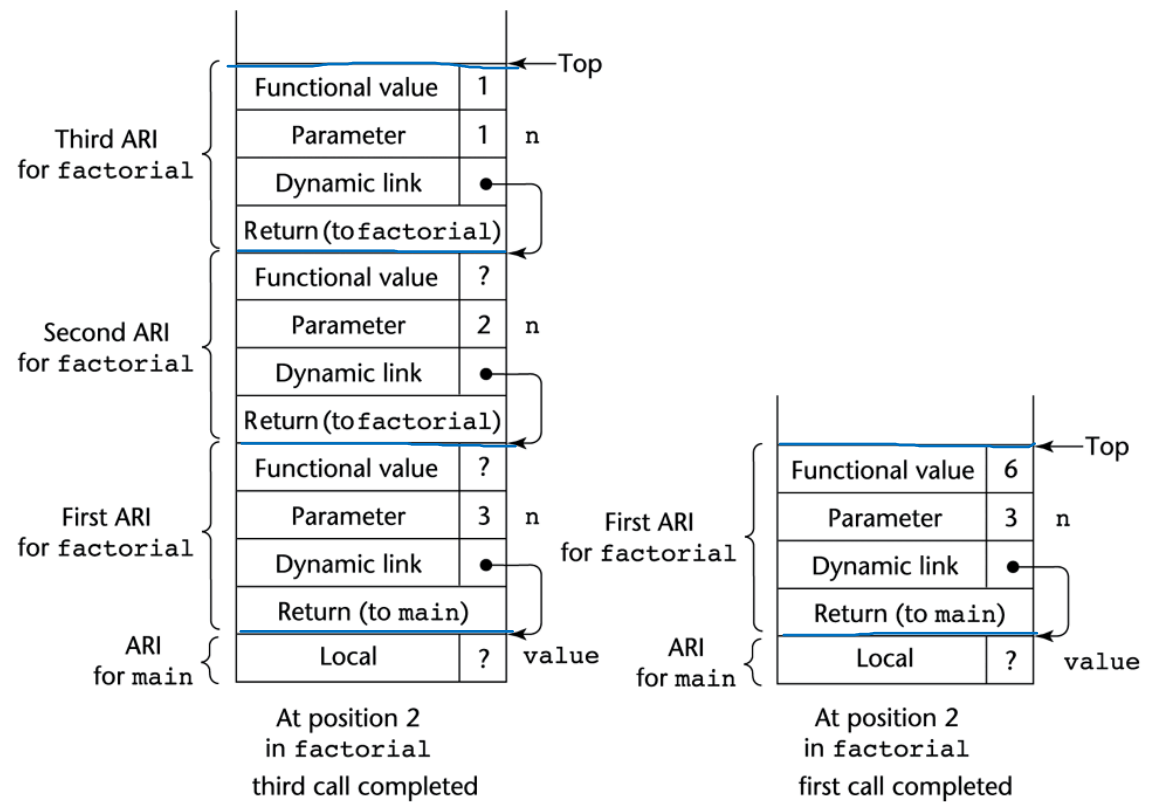


ARI = activation record instance

Esempio ricorsivo in C

```
int factorial(int n)
{
    if(n<=1) return 1;
    else return
        n*factorial(n-1);
}

void main( )
{
    int value;
    value = factorial(3);
}
```



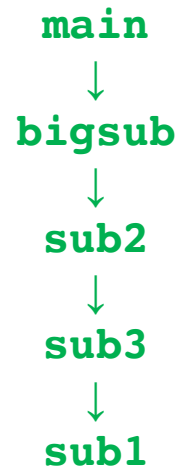
ARI = activation record instance

Esempio con funzioni annidate in JavaScript

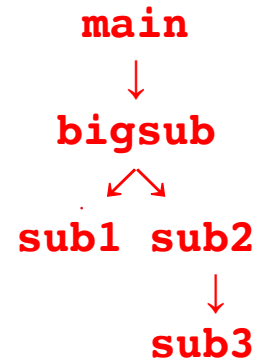
La sequenza delle
chiamate:

1. main chiama bigsub
2. bigsub chiama sub2
3. sub2 chiama sub3
4. sub3 chiama sub1

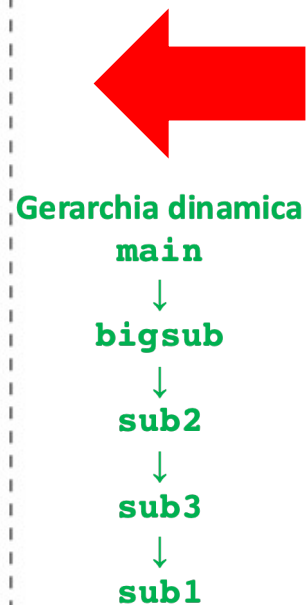
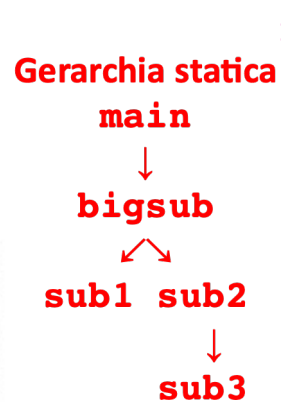
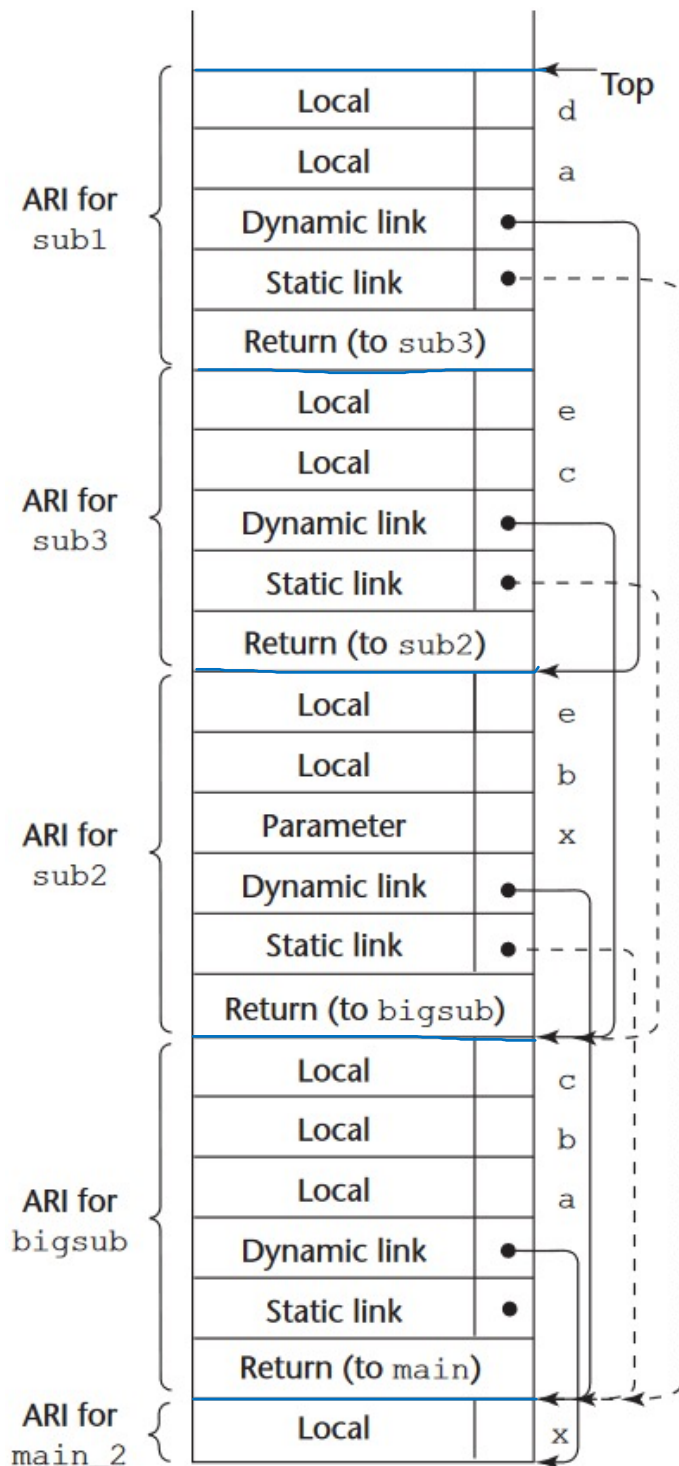
Gerarchia dinamica



Gerarchia statica

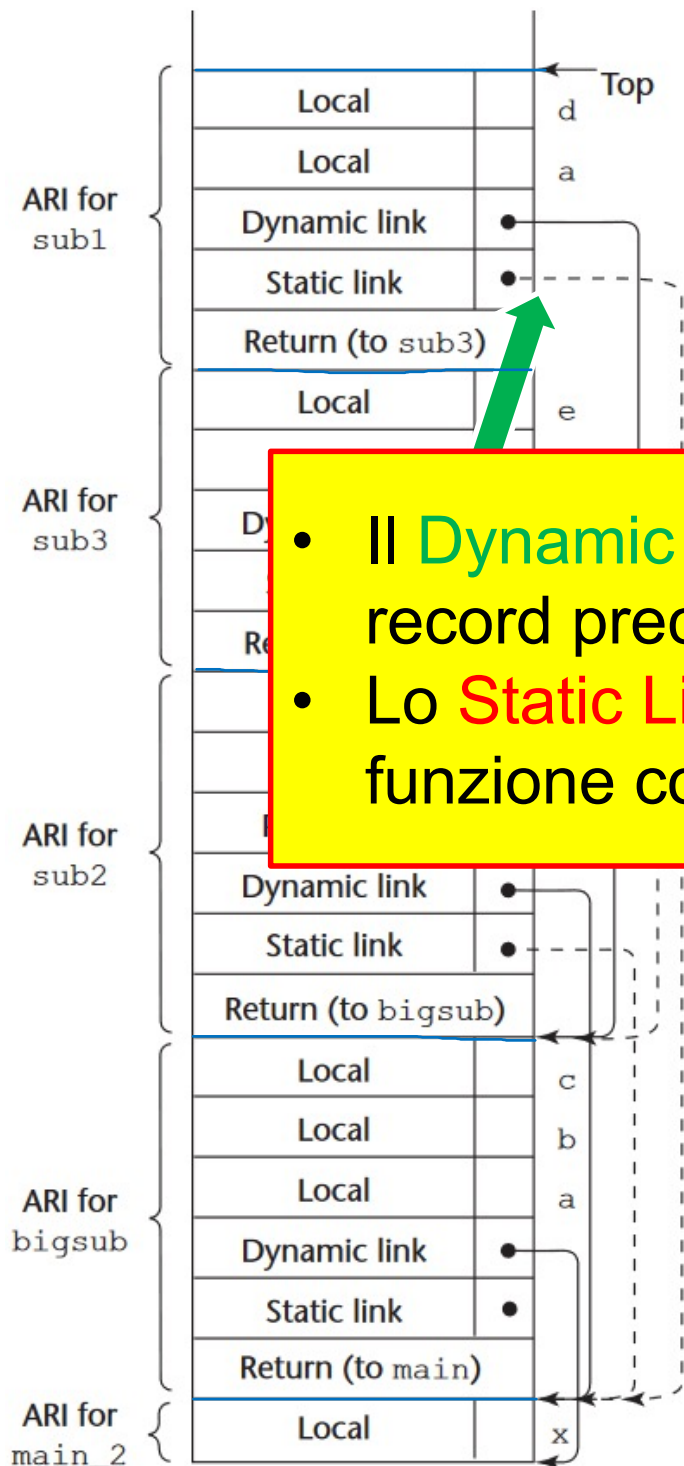


```
function main(){  
  var x;  
  function bigsub() {  
    var a, b, c;  
    function sub1 {  
      var a, d;  
      a = b + c; <----- eseguito 1^  
    } // end of sub1  
    function sub2(x) {  
      var b, e;  
      function sub3() {  
        var c, e;  
        sub1();  
        e = b + a; <-- eseguito 2^  
      } // end of sub3  
      sub3();  
      a = d + e; <----- eseguito 3^  
    } // end of sub2  
    sub2(7);  
  } // end of bigsub  
  bigsub();  
} // end of main
```

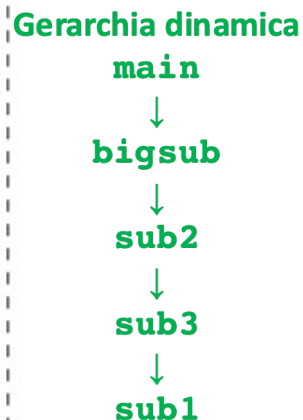
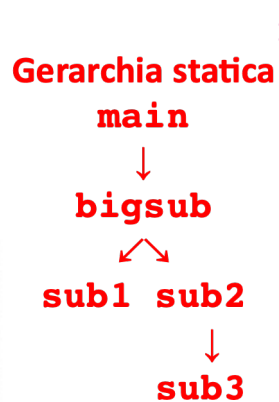


```

function main(){
  var x;
  function bigsub() {
    var a, b, c;
    function sub1 {
      var a, d;
      a = b + c; <---- eseguito 1^
    } // end of sub1
    function sub2(x) {
      var b, e;
      function sub3() {
        var c, e;
        sub1();
        e = b + a; <-- eseguito 2^
      } // end of sub3
      sub3();
      a = d + e; <---- eseguito 3^
    } // end of sub2
    sub2(7);
  } // end of bigsub
  bigsub();
} // end of main
  
```

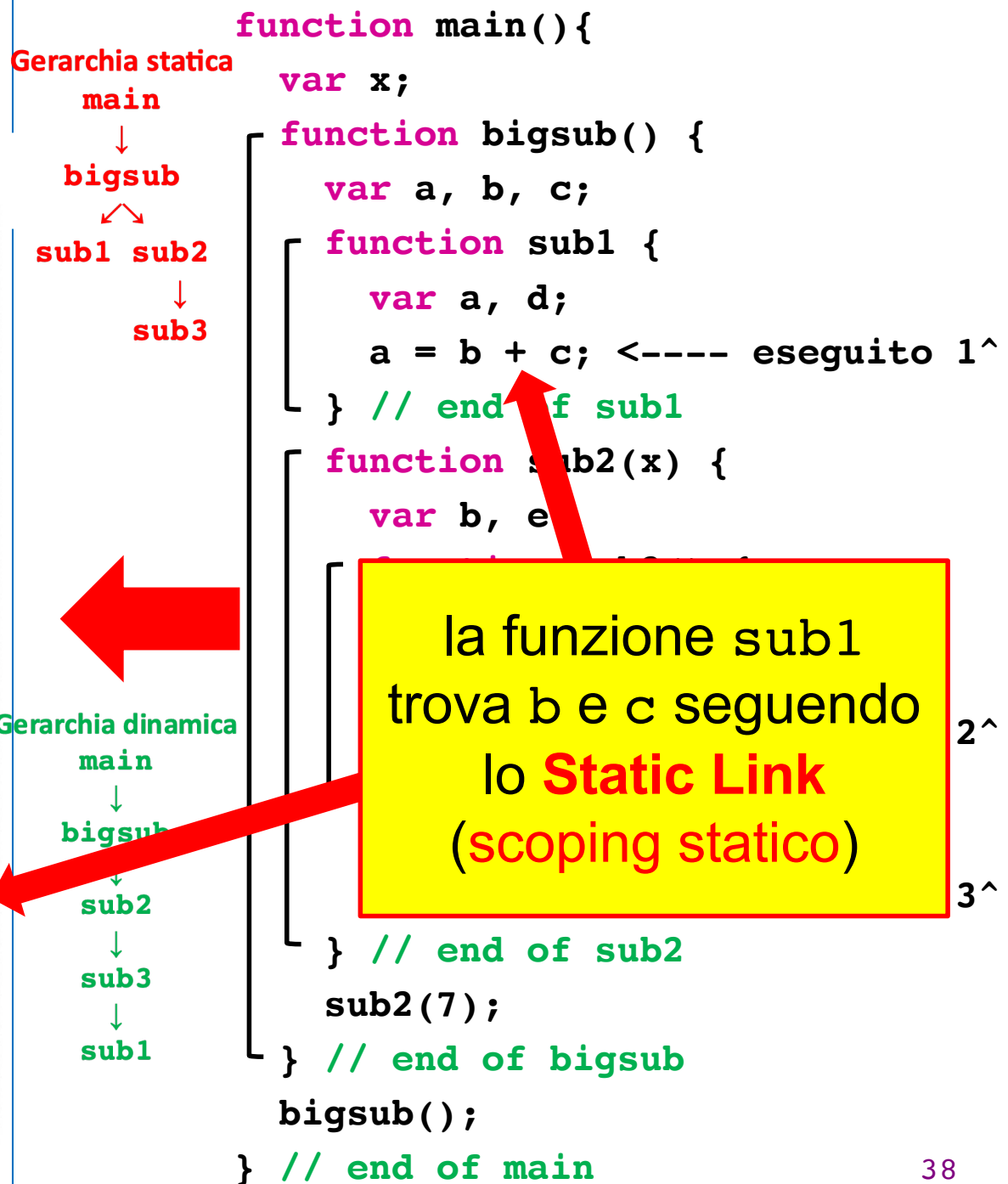
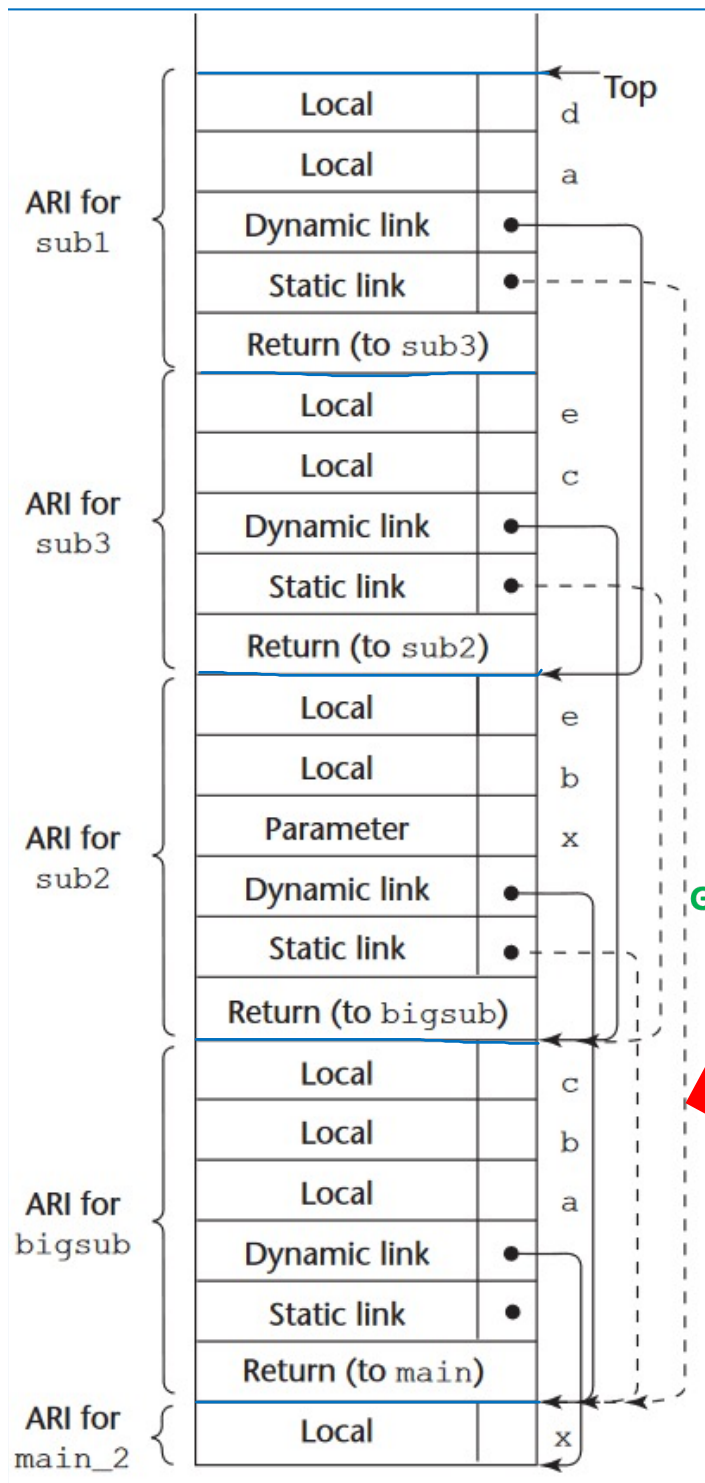


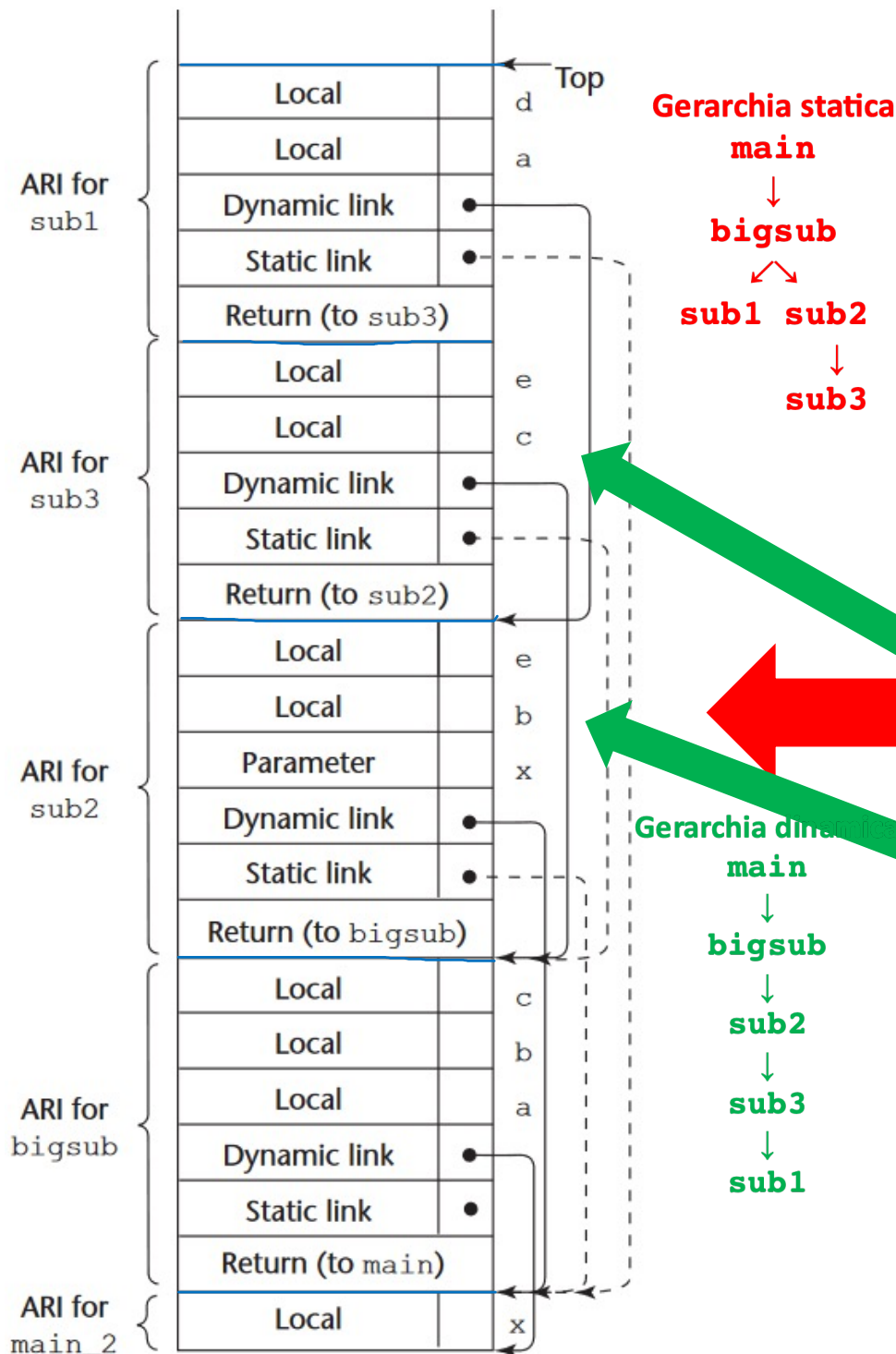
- Il **Dynamic Link** punta al record precedente
- Lo **Static Link** punta alla funzione contenitrice



```

function main(){
  var x;
  function bigsub() {
    var a, b, c;
    function sub1 {
      var a, d;
      a = b + c; <---- eseguito 1^
    } // end of sub1
    function sub2(x) {
      var b, e;
      function sub3() {
        var c, e;
        sub1();
        e = b + a; <-- eseguito 2^
      } // end of sub3
      sub3();
      a = d + e; <---- eseguito 3^
    } // end of sub2
    sub2(7);
  } // end of bigsub
  bigsub();
} // end of main
  
```



```
function main(){
```

```
  var x;
```

```
  function bigsub() {
```

```
    var a, b, c;
```

```
    function sub1 {
```

```
      var a, d;
```

```
      a = b + c; <----- eseguito 1^
```

```
    } // end of sub1
```

```
    function sub2(x) {
```

```
      var b, e;
```

Nei linguaggi con **scoping dinamico** (non JavaScript)

non c'è lo Static Link: le variabili non locali si cercano scorrendo la pila, ossia seguendo il **Dynamic Link** (Trova altri b e c)

```
  } // end of main
```

Implementazione di un ambiente in OCaml

Premessa

- Vediamo come implementare in OCaml un **ambiente minimale**
 - no blocchi
 - no record di attivazione
 - solo una "pila" di binding

Ambiente

- Un **ambiente** Σ è una **collezione di binding**
- Ad esempio: $\Sigma = \{x \rightarrow 25, y \rightarrow 7\}$
- L'ambiente Σ contiene due “binding”:
 - l'associazione tra l'identificatore **x** e il valore **25**
 - l'associazione tra l'identificatore **y** e il valore **7**
 - mentre l'identificatore **z** non è legato nell'ambiente
- Astrattamente un ambiente è una **funzione** di tipo
 $\Sigma: \text{Ide} \rightarrow \text{Value} + \text{Unbound}$
- L'uso della costante **Unbound** permette di rendere la funzione totale

Ambiente

- Dato un ambiente Σ : **Ide \rightarrow Value + Unbound**
- $\Sigma(\mathbf{x})$ denota il valore \mathbf{v} associato a \mathbf{x} nell'ambiente oppure il valore speciale **Unbound**
- $\Sigma[\mathbf{x}=\mathbf{v}]$ indica l'ambiente **esteso** così definite

$$\Sigma[\mathbf{x}=\mathbf{v}](\mathbf{y}) = \begin{cases} \mathbf{v} & \text{se } \mathbf{y} = \mathbf{x} \\ \Sigma(\mathbf{y}) & \text{se } \mathbf{y} \neq \mathbf{x} \end{cases}$$

- Ad esempio: se $\Sigma = \{x \rightarrow 25, y \rightarrow 7\}$ allora
 $\Sigma[x=5] = \{x \rightarrow 5, y \rightarrow 7\}$

Implementazione ambiente (semplice, come lista di coppie)



```
(* ambiente vuoto *)
let emptyenv = []

(* aggiornamento ambiente s con associazione
(x,v) *)

let bind s x v = (x, v)::s

(* operazione di referencing in un ambiente s *)
let rec lookup s x =
  match s with
  | [] -> failwith ("not found")
  | (y, v)::r -> if x = y then v else
                  lookup r x
```

Implementazione alternativa (come funzione polimorfa)



Un **ambiente** è una funzione che associa a ogni **identificatore** (**ide**) un **valore di tipo** **'t**, dove 't rappresenta i valori "esprimibili" nel linguaggio, cioè quei valori che possono essere calcolati durante la valutazione delle espressioni

$$\Sigma: \text{Ide} \rightarrow \text{Value} + \text{Unbound}$$

```
(* ambiente polimorfo *)  
type 't env = ide -> 't      (* 't sarà il tipo dei  
                               valori esprimibili *)
```

Serve un **ambiente vuoto** come punto di partenza per le valutazioni. **emptyenv** è una funzione che prende un identificatore (**x**) e restituisce il valore speciale **UnBound**, indicando che l'ambiente non ha alcun legame iniziale

$$[](\text{x}) = \text{Unbound}$$

```
(* ambiente vuoto *)  
let emptyenv = fun x -> UnBound  (* valore speciale *)
```


Aggiornamento ambiente

- L'operazione di **referencing** $s x$ rappresenta la ricerca del **valore associato a un identificatore** x nell'ambiente s
- La funzione **bind** aggiorna l'ambiente s associando un identificatore x a un valore v

```
(* operazione di referencing in un ambiente s *)  
s x
```

```
(* aggiornamento ambiente s con associazione (x,v) *)  
let bind s x v =  
  fun i -> if (i = x) then v else (s i)
```

bind restituisce una nuova funzione (ambiente) che:

- associa x al valore v
- per ogni altro identificatore $i \neq x$,
usa l'ambiente originale s

$$\Sigma[x=v](y) = \begin{cases} v & \text{se } y = x \\ \Sigma(y) & \text{se } y \neq x \end{cases}$$

Implementazione ambiente



```
(* ambiente polimorfo *)
type 't env = ide -> 't          (* 't sarà il tipo dei
                                   valori esprimibili *)

(* operazione di referencing in un ambiente s *)
s x

(* ambiente vuoto *)
let emptyenv = fun x -> UnBound  (* valore speciale *)

(* aggiornamento ambiente s con associazione (x,v) *)
let bind s x v =
  fun i -> if (i = x) then v else (s i)
```

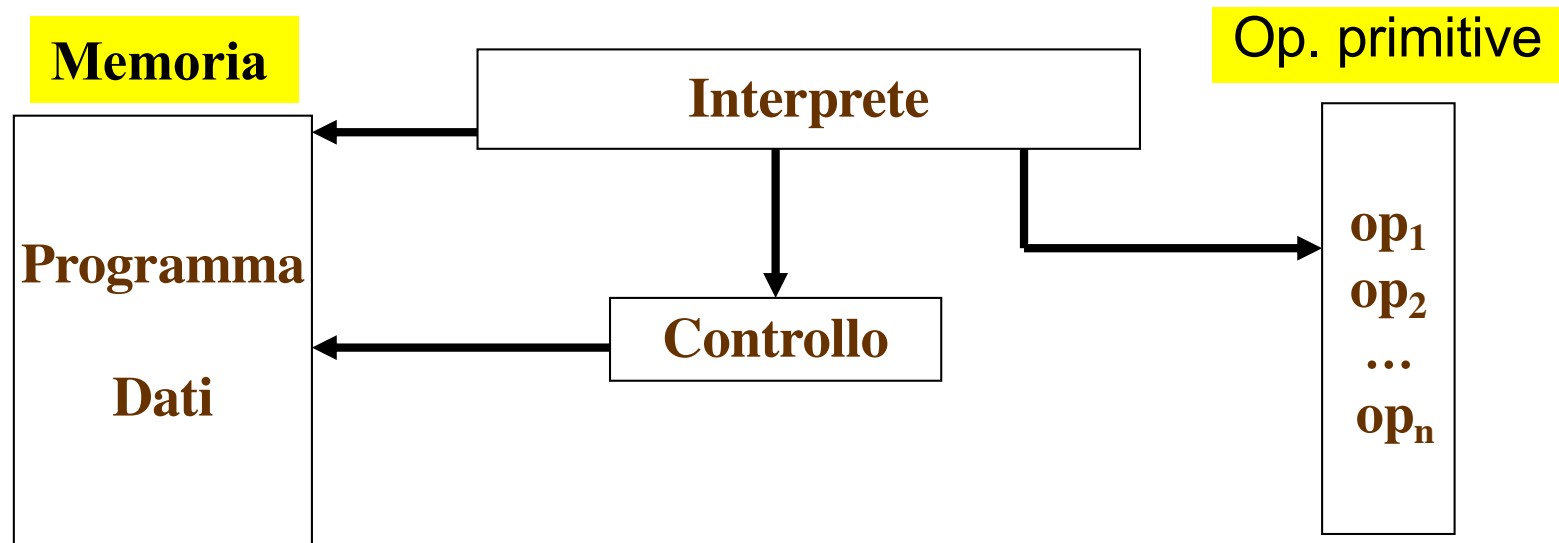
Linguaggio MiniCaml

Linguaggio funzionale didattico

- Consideriamo il **nucleo di un linguaggio funzionale**
 - sottoinsieme di ML, senza tipi né pattern matching
- **Obiettivo:**
 - esaminare tutti gli **aspetti** relativi alla **implementazione** dell'interprete, e
 - del **supporto a run time** per il linguaggio

Struttura dell'interprete

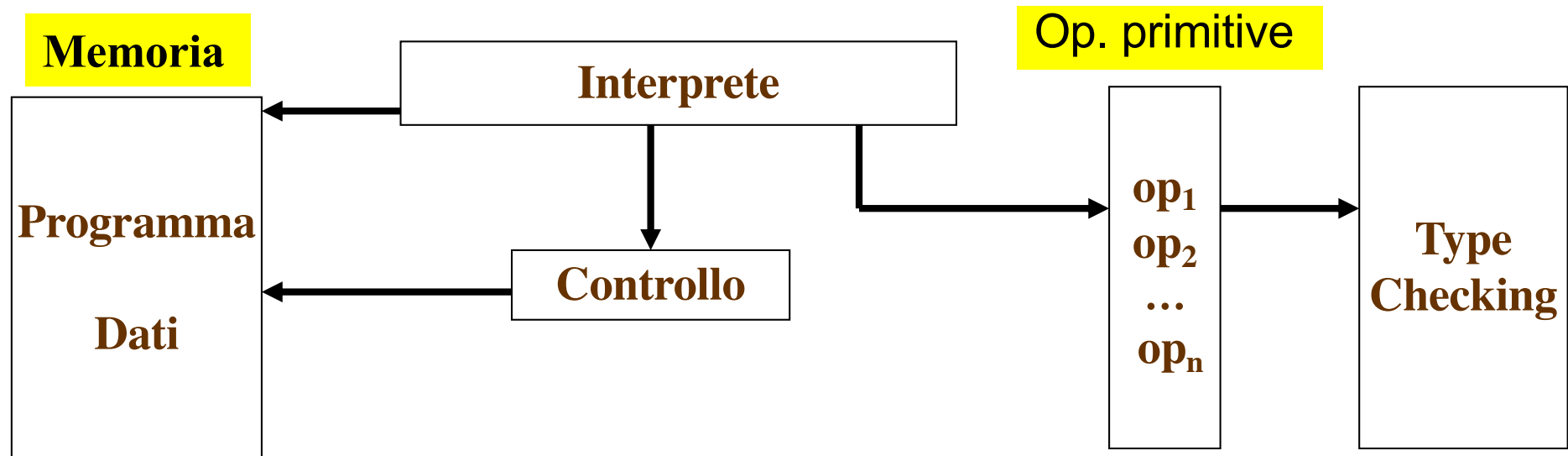
Seguiremo la struttura generale delle macchine astratte



Struttura dell'interprete

Seguiremo la struttura generale delle macchine astratte

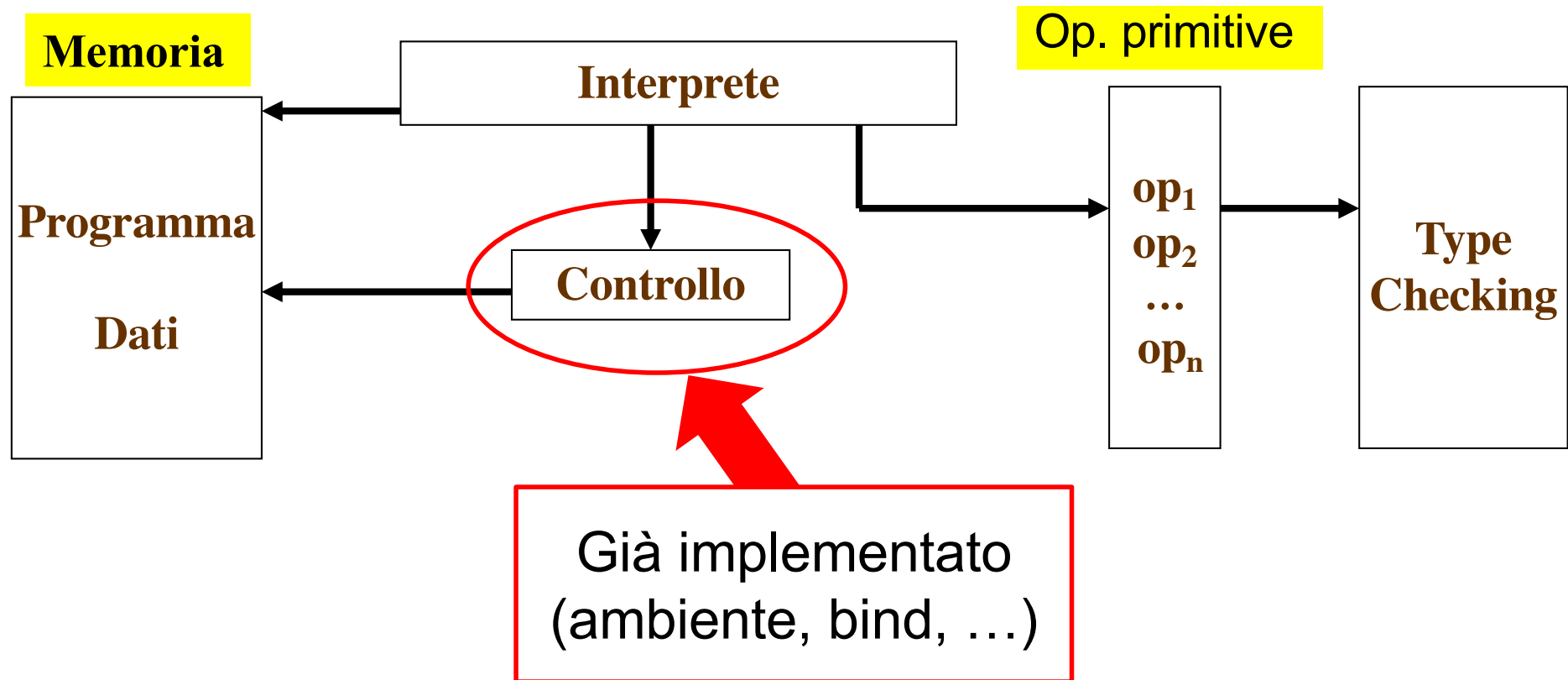
- Esplicitando anche le operazioni di **type checking (dinamico)**



Struttura dell'interprete

Seguiremo la struttura generale delle macchine astratte

- Esplicitando anche le operazioni di **type checking (dinamico)**

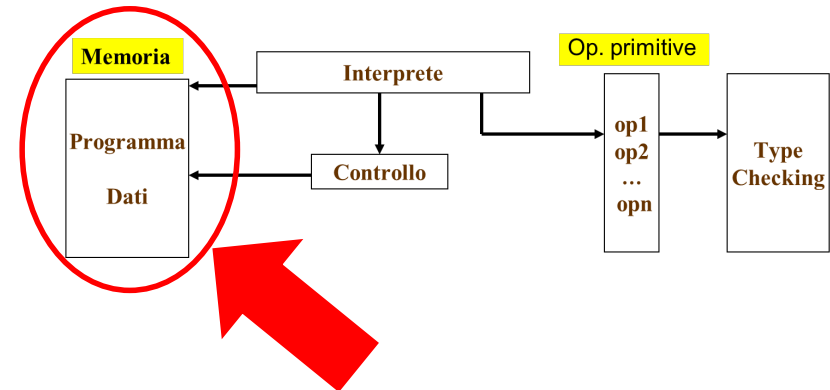


Linguaggio funzionale didattico

type ide = string

type exp =

CstInt of int	(* Letterale n *)
CstTrue	(* Letterale true *)
CstFalse	(* Letterale false *)
Sum of exp * exp	(* Somma *)
Diff of exp * exp	(* Sottrazione *)
Prod of exp * exp	(* Prodotto *)
Div of exp * exp	(* Divisione *)
Eq of exp * exp	(* Uguale *)
Iszero of exp	(* Controlla se uguale a zero *)
Or of exp * exp	(* Or logico *)
And of exp * exp	(* And logico *)
Not of exp	(* Not logico *)
Ifthenelse of exp * exp * exp	(* Espressione condizionale *)
Den of ide	(* Entità denotabile (variabile) *)
Let of ide * exp * exp	(* Dichiarazione di ide: modifica ambiente *)
Fun of ide * exp	(* Astrazione di funzione (non ricorsiva, con singolo parametro) *)
Apply of exp * exp	(* Applicazione di funzione *)
Letrec of ide * ide * exp * exp	(* Dichiarazione di funzione ricorsiva (con singolo parametro) *)



La parte semplice: espressioni

```
type exp =  
  | CstInt of int  
  | CstTrue  
  | CstFalse  
  | Sum of exp * exp  
  | Diff of exp * exp  
  | Prod of exp * exp  
  | Div of exp * exp  
  | Eq of exp * exp  
  | Iszero of exp  
  | Or of exp * exp  
  | And of exp * exp  
  | Not of exp  
  | Ifthenelse of exp * exp * exp
```

Ciclo interprete

let rec eval (e: exp) =
match e **with**

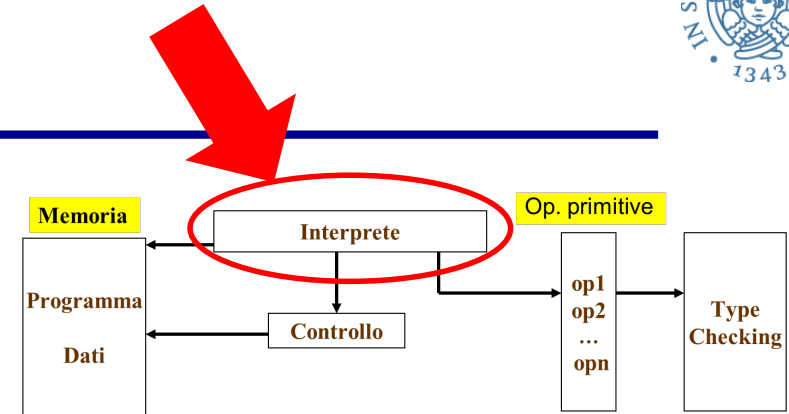
| CstInt(n) -> Int(n)

| CstTrue -> Bool(true)

| CstFalse -> Bool(false)

| Iszero(e1) -> ???

| Den(i) -> ???



DATI PRIMITIVI
INT e BOOL

Valori esprimibili

- **Valori esprimibili** (evaluation types): valori che possono essere calcolati e restituiti durante la valutazione di espressioni

```
type evT = Int of int
          | Bool of bool
          | Unbound
          | ...
```

- **Tipi**

```
(* tipi di dato esistenti *)
type tname =
  | TInt
  | TBool
  | ...
```

Ambiente

- **Ambiente:** associazione **ide** **evT** **env**

```
type 't env = ide -> 't
(* 't sarà il tipo dei valori esprimibili *)
```

```
let bind (s: evT env) (x: ide) (v: evT) =
  fun i -> if (i = x) then v else s i
```

$$\Sigma[x=v](y) = \begin{cases} v & \text{se } y = x \\ \Sigma(y) & \text{se } y \neq x \end{cases}$$

Come determinare il tipo

- Nell'ambiente di un interprete, i valori devono avere anche l'informazione sul tipo per consentire il **type checking dinamico**
- La funzione **getType** prende un valore **x** di tipo **evT** e restituisce un valore di tipo **tname** che rappresenta il tipo. Restituisce quindi il tipo (**descrittore**) di un valore esprimibile **evT** di **x**

```
(* associa a ogni valore il suo  
descrittore di tipo *)  
let getType (x: evT) : tname =  
  match x with  
  | Int(n)  -> TInt  
  | Bool(b) -> TBool  
  | ...
```



Typechecking (dinamico)

La funzione **typecheck** prende una coppia (type, typeDescriptor) in cui:

- **type** è un valore di tipo **tname**, cioè il **tipo atteso** del valore.
- **typeDescriptor** è un valore di tipo **evT**, il **valore esprimibile da verificare**

let typecheck (type, typeDescriptor) =
match type with

| TInt ->

(match typeDescriptor with

| Int(u) -> true

| _ -> false)

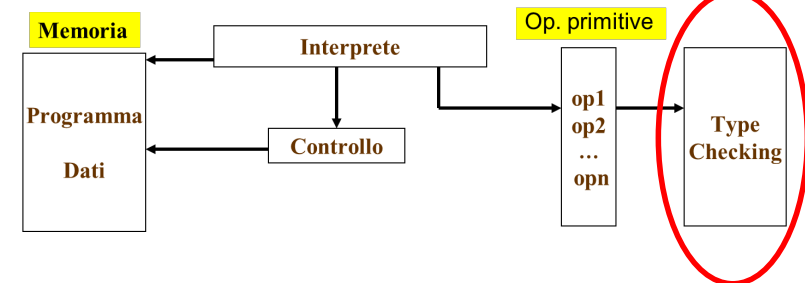
| TBool ->

(match typeDescriptor with

| Bool(u) -> true

| _ -> false)

| _ -> failwith ("not a valid type");;

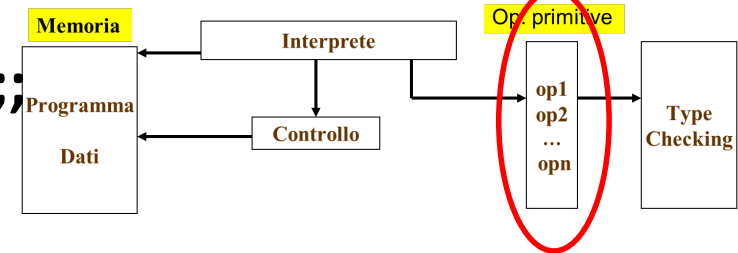


Se **type** è **TInt**, **typecheck** controlla
se **typeDescriptor** è di tipo **Int**

val typecheck : tname * evT -> bool = <fun>

Operazioni di base

```
let is_zero x = match (typecheck(TInt,x), x) with
  | (true, Int(y)) -> Bool(y=0)
  | (_, _) -> failwith("run-time error");;
```



```
let int_eq(x,y) =
  match (typecheck(TInt,x), typecheck(TInt,y), x, y) with
  | (true, true, Int(v), Int(w)) -> Bool(v = w)
  | (_,_,_,_) -> failwith("run-time error ")...
```

```
let int_plus(x, y) =
  match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
  | (true, true, Int(v), Int(w)) -> Int(v + w)
  | (_,_,_,_) -> failwith("run-time error ");;
```

Queste funzioni sono implementate
con controlli di tipo dinamici
Si applicano a valori già valutati

Operazioni di base

Implementazione operazioni di base



```
let is_zero x = match (typecheck(TInt,x), x) with
  | (true, Int(y)) -> Bool(y=0)
  | (_, _) -> failwith("run-time error");;
```

```
let int_eq(x,y) =
  match (typecheck(TInt,
    | (true, true, Int(v), Int(w)) -> failwith("run-time error");;
```

Controlla se l'argomento **x** (di tipo **evt**) è zero, ammesso che l'argomento sia intero. Equivale a

```
let is_zero(x) = if typecheck(TInt, x)
then match x with
  | Int(y) -> Bool(y = 0)
  | _ -> failwith("run-time error");;
```

```
let int_plus(x, y) =
  match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
  | (true, true, Int(v), Int(w)) -> Int(v + w)
  | (_,_,_,_) -> failwith("run-time error ");;
```


Operazioni di base

```
let is_zero x = match (typecheck(TInt,x), x) with  
  | (true, Int(y)) -> Int(y)  
  | (_, _) -> failwith("run-time error");;
```

```
let int_eq(x,y) =  
  match (typecheck(TInt,x), typecheck(TInt,y)) with  
  | (true, true) -> Int(0)  
  | (_,_,_,_) -> failwith("run-time error");;
```

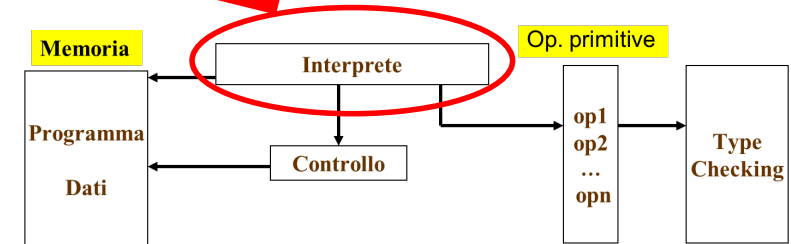
```
let int_plus(x,y) =  
  match (typecheck(TInt,x), typecheck(TInt,y)) with  
  | (true, true, Int(v), Int(w)) -> Int(v + w)  
  | (_,_,_,_) -> failwith("run-time error");;
```

Le operazioni di base sono implementate tramite una regola di **valutazione eager**:
prima di applicare l'operatore, si valutano tutti i sottoalberi (sotto-espressioni)

Ciclo interprete

let rec eval (e:exp) (s: evT env) : evT =
match e **with**

- | CstInt(n) -> Int(n)
- | CstTrue -> Bool(true)
- | CstFalse -> Bool(false)
- | Iszero(e1) -> is_zero(eval e1 s)
- | Eq(e1, e2) -> int_eq((eval e1 s), (eval e2 s))
- | Sum(e1, e2) -> int_plus ((eval e1 s), (eval e2 s))
- | Diff(e1, e2) -> int_sub ((eval e1 s), (eval e2 s))
- | Prod(e1,e2) -> int_times((eval e1 s), (eval e2 s))
- | Div(e1,e2) -> int_div((eval e1 s), (eval e2 s))
- | And(e1, e2) -> bool_and((eval e1 s), (eval e2 s))
- | Or(e1, e2) -> bool_or ((eval e1 s), (eval e2 s))
- | Not(e1) -> bool_not((eval e1 s))



$$\frac{\Sigma \triangleright e_1 \Rightarrow v_1 \quad \Sigma \triangleright e_2 \Rightarrow v_2}{\Sigma \triangleright Op(e_1, e_2) \Rightarrow op(v_1, v_2)}$$

Condizionale: regole operazionali

SINTASSI ASTRATTA

Ifthenelse of exp * exp * exp

Regole semantica operazionale

Regole operazionali

$$\frac{\Sigma \triangleright cond \Rightarrow true \quad \Sigma \triangleright e_1 \Rightarrow v_1}{\Sigma \triangleright Ifthenelse(cond, e_1, e_2) \Rightarrow v_1}$$

$$\frac{\Sigma \triangleright cond \Rightarrow false \quad \Sigma \triangleright e_2 \Rightarrow v_2}{\Sigma \triangleright Ifthenelse(cond, e_1, e_2) \Rightarrow v_2}$$

Condizionale: regola interprete

...

```
| Ifthenelse (cond,e1,e2) ->  
  let g = eval cond s in  
  match (typecheck(TBool, g), g) with  
  | (true, Bool(true)) -> eval e1 s  
  | (true, Bool(false)) -> eval e2 s  
  | (_, _) -> failwith ("non boolean guard")
```

Solo la valutazione del condizionale **non** segue una strategia **eager**: la **valutazione dei sottoalberi** avviene in base alla valutazione della **guardia**

$$\frac{\Sigma \triangleright \text{cond} \Rightarrow \text{true} \quad \Sigma \triangleright e_i \Rightarrow v_i}{\Sigma \triangleright \text{Ifthenelse}(\text{cond}, e_1, e_2) \Rightarrow v_i}$$

Binding identificatori

SINTASSI ASTRATTA

Den of ide

Regola semantica operativa

$$\Sigma \triangleright \text{Den}(i) \Rightarrow \Sigma(i)$$

Regola Interprete (nella funzione eval)

Den(i) valuta un identificatore cercandone il valore nell'ambiente s

let rec eval (e:exp) (s: evT env) : evT =

match e with

| ...

| **Den(i) ->** $s\ i$

L'espressione OCaml x
diventa

$\text{Den}("x")$

Esempio

L'espressione $x+1$ diventa $\text{Sum}(\text{Den}("x"), \text{CstInt}(1))$

let rec eval (e:exp) (s: evT env) : evT =

match e **with**

| CstInt(n) -> Int(n)

| ...

| Sum(e1, e2) -> int_plus ((eval e1 s), (eval e2 s))

| ...

| Den(i) -> s i

| ...

$$\Sigma \triangleright \text{Den}(i) \Rightarrow \Sigma(i)$$

$$\frac{\Sigma \triangleright e_1 \Rightarrow v_1 \quad \Sigma \triangleright e_2 \Rightarrow v_2}{\Sigma \triangleright \text{Sum}(e_1, e_2) \Rightarrow \text{int_plus}(v_1, v_2)}$$

$$\frac{s \vdash \text{Den}("x") \Rightarrow \text{Int}(2) \quad s \vdash \text{CstInt}(1) \Rightarrow \text{Int}(1)}{s \vdash \text{Sum}(\text{Den}("x"), \text{CstInt}(1)) \Rightarrow \text{Int}(3)}$$

Blocco: Let(x, e1, e2)

- Con il **Let** possiamo **cambiare l'ambiente in punti arbitrari** all'interno di una espressione
 - facendo sì che l'ambiente "nuovo" valga soltanto durante la valutazione del "corpo del blocco", l'espressione **e2**
 - lo stesso nome può denotare entità distinte in blocchi diversi
- I **blocchi** possono essere **annidati**
 - e l'ambiente locale di un blocco più esterno può essere visibile e utilizzabile nel blocco più interno
 - ✓ come ambiente non locale!
- Il blocco
 - porta naturalmente a una semplice **gestione dinamica della memoria locale** (**stack** dei record di attivazione)
 - si sposa naturalmente con la regola di **scoping statico**
 - ✓ per la gestione dell'**ambiente non locale**

Semantica operativa del blocco

SINTASSI ASTRATTA

Let of ide * exp * exp

Regola semantica operativa

$$\frac{\Sigma \triangleright e_1 \Rightarrow v_1 \quad \Sigma[x = v_1] \triangleright e_2 \Rightarrow v_2}{\Sigma \triangleright \text{Let}(x, e_1, e_2) \Rightarrow v_2}$$

La regola dell'interprete

let rec eval (e:exp) (s: evT env) : evT =
match e **with**
| ...
| Let(i, e, ebody) ->
 eval ebody (**bind s i (eval e s)**)

L'espressione **ebody** (il corpo del blocco) è valutata nell'**ambiente corrente "esterno"**, esteso con l'associazione tra il nome **i** e il valore calcolato di **e**

$$\frac{\Sigma \triangleright e \Rightarrow v_1 \quad \Sigma[i = v_1] \triangleright ebody \Rightarrow v_2}{\Sigma \triangleright \text{Let}(i, e, ebody) \Rightarrow v_2}$$

Esempio

L'espressione OCaml `let x = 2 in x+1` diventa

`Let ("x", CstInt(2), Sum(Den("x"), CstInt(1)))`

$$\frac{\Sigma \triangleright e \Rightarrow v_1 \quad \Sigma[i = v_1] \triangleright ebody \Rightarrow v_2}{\Sigma \triangleright Let(i, e, ebody) \Rightarrow v_2}$$

$$\frac{\begin{array}{c} \emptyset \vdash CstInt(2) \Rightarrow Int(2) \quad \frac{\emptyset[x = Int(2)] \vdash Den("x") \Rightarrow Int(2) \quad \emptyset[x = Int(2)] \vdash CstInt(1) \Rightarrow Int(1)}{\emptyset[x = Int(2)] \vdash Sum(Den("x"), CstInt(1)) \Rightarrow Int(3)} \\ \emptyset \vdash let ("x", CstInt(2), Sum(Den("x"), CstInt(1))) \Rightarrow Int(3) \end{array}}$$

Semantica operativa del blocco

Σ = run-time stack

push RA su Σ

$$\frac{\Sigma \triangleright e_1 \Rightarrow v_1 \quad \Sigma[x = v_1] \triangleright e_2 \Rightarrow v_2}{\Sigma \triangleright \text{Let}(x, e_1, e_2) \Rightarrow v_2}$$

Uscita blocco
pop su Σ

REPL

```
# let myp =
```

```
  Let("x", CstInt(30), Let("y", CstInt(12), Sum(Den("x"),Den("y"))));;
```

```
val myp : exp = Let ("x", CstInt 30, Let ("y", CstInt 12, Sum (Den "x", Den "y")))
```

```
# eval myp emptyEnv;;
```

```
- : evT = Int 42
```

```
let x = 30 in let y = 12 in x+y
```

```
# let myp' = CstInt(3);;
```

```
val myp' : exp = CstInt 3
```

```
# let e = Eq(CstInt(5),CstInt(5));;
```

```
val e : exp = Eq (CstInt 5, CstInt 5)
```

```
let e = (5 = 5)
```

```
# let myite = Ifthenelse(e,myp,myp');;
```

```
val myite : exp =
```

```
Ifthenelse (Eq (CstInt 5, CstInt 5), Let ("x", CstInt 30, Let ("y", CstInt 12, Sum (Den "x",  
Den "y"))), CstInt 3)
```

```
# eval myite emptyEnv;;
```

```
- : evT = Int 42
```

```
if (5 = 5) then  
  let x = 30 in let y = 12 in x+y  
else 3
```

Funzioni

- **astrazione funzionale**
 - **Fun of ide * exp**
- **applicazione di funzione**
 - **Apply of exp * exp**

Astrazione funzionale

- Funzioni anonime
 - **Fun**("**x**" , **fbody**)
 - "**x**" **parametro formale**,
 - **fbody** **corpo della funzione**,

L'espressione OCaml

let f x = x+7 in f 2 equivalente a **let f = (fun x -> x+7) in f 2**

diventa

Let("f", Fun("x", Sum(Den("x"), CstInt(7))), Apply(Den("f"),CstInt(2)))

Semplificazione sintattica

- Per semplicità assumiamo che l'applicazione funzionale sia del **primo ordine**
 - Il primo argomento dell'applicazione funzionale deve essere il **nome** della funzione da invocare
 - **Apply**(**e**, **arg**) deve avere la forma **Apply**(**Den**("f"), **arg**)
- Inoltre, per ora **non** consideriamo funzioni ricorsive

Funzioni

SINTASSI ASTRATTA

Fun of ide * exp

Apply of Den("f") * exp

- Identificatori (**parametri formali**) nel costrutto di astrazione

Fun of ide * exp

- Espressioni (**parametri attuali**) nel costrutto di applicazione

Apply of Den("f") * exp

- Per ora
 - **non** trattiamo la **modalità di passaggio parametri**:
si valutano le espressioni corrispondenti ai parametri attuali
e si legano i valori ottenuti, nell'ambiente, ai rispettivi
parametri formali
 - **non** trattiamo le **funzioni ricorsive**
 - assumiamo di avere **funzione unarie**

Con le funzioni, il linguaggio funzionale è completo (un linguaggio funzionale reale (tipo ML) ha in più: tipi, pattern matching, eccezioni) 77

Analisi semantica

- Come estendere i tipi esprimibili (**evT**) per comprendere le astrazioni funzionali?
 - Qual è il **valore di una funzione**?
- Assumiamo **scoping statico** (vedremo poi quello **dinamico**): i riferimenti non locali dell'astrazione sono risolti nell'**ambiente di dichiarazione della funzione**

```
type evT = | Int of int | Bool of bool | Unbound
           | Closure of ide * exp * evT env
```

- La definizione mostra che il valore esprimibile di una astrazione funzionale è una **chiusura**, che comprende:
 - il **nome** del **parametro formale** (**ide**)
 - il **corpo** della **funzione** dichiarata (**exp**)
 - l'**ambiente al momento della dichiarazione** (**evT env**)

Semantica operativa dell'astrazione (scoping statico)

La valutazione di un'astrazione funzionale restituisce una **chiusura** che include:

- il nome del **parametro formale** ("**x**")
- il **codice** della **funzione** dichiarata (**e**)
- l'**ambiente al momento della dichiarazione** (**Σ**)

$$\Sigma \triangleright Fun(x, e) \Rightarrow Closure("x", e, \Sigma)$$

Semantica operativa dell'applicazione di funzione (scoping statico)

- Si valutano le espressioni corrispondenti ai parametri attuali (**arg**)
- si legano i valori ottenuti (**va**) ai rispettivi parametri formali (**x**), nell'ambiente di dichiarazione della funzione (Σ_{fDecl}), presente all'interno della chiusura (**Closure("x", body, fDecl)**)
- si valuta il corpo della funzione (**body**) nell'ambiente per l'applicazione così ottenuto

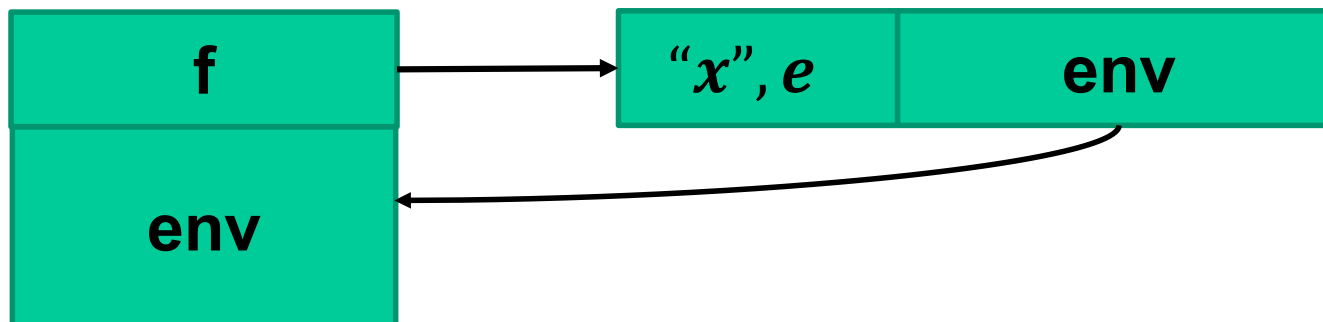
$$\begin{array}{l}
 \Sigma \triangleright Den("f") \Rightarrow Closure("x", body, \Sigma_{fDecl}) \\
 \Sigma \triangleright arg \Rightarrow va \quad \Sigma_{fDecl}[x = va] \triangleright body \Rightarrow v \\
 \hline
 \Sigma \triangleright Apply(Den("f"), arg) \Rightarrow v
 \end{array}$$

Approccio **call-by-value**

Dichiarazione di una funzione

$$\frac{\begin{array}{l} \Sigma \triangleright Fun(x, e) \Rightarrow Closure("x", e, \Sigma) \\ \Sigma[f = Closure("x", e, \Sigma)] \triangleright e' \Rightarrow v' \end{array}}{\Sigma \triangleright Let("f", Fun(x, e), e') \Rightarrow v'}$$

La valutazione di una dichiarazione di funzione con il **Let** crea una **chiusura** che cattura l'ambiente corrente al momento della dichiarazione e che viene usata per estendere l'ambiente nel quale valutare un'espressione



Dichiarazione di una funzione: interprete

$$\frac{\begin{array}{l} \Sigma \triangleright Fun(x, e) \Rightarrow Closure("x", e, \Sigma) \\ \Sigma[f = Closure("x", e, \Sigma)] \triangleright e' \Rightarrow v' \end{array}}{\Sigma \triangleright Let("f", Fun(x, e), e') \Rightarrow v'}$$

let rec eval (e:exp) (s: evT env) : evT =

match e **with**

| ...

| Let(f, e, ebody) ->

eval ebody (bind s i (eval e s))

Fun(x, fbody)

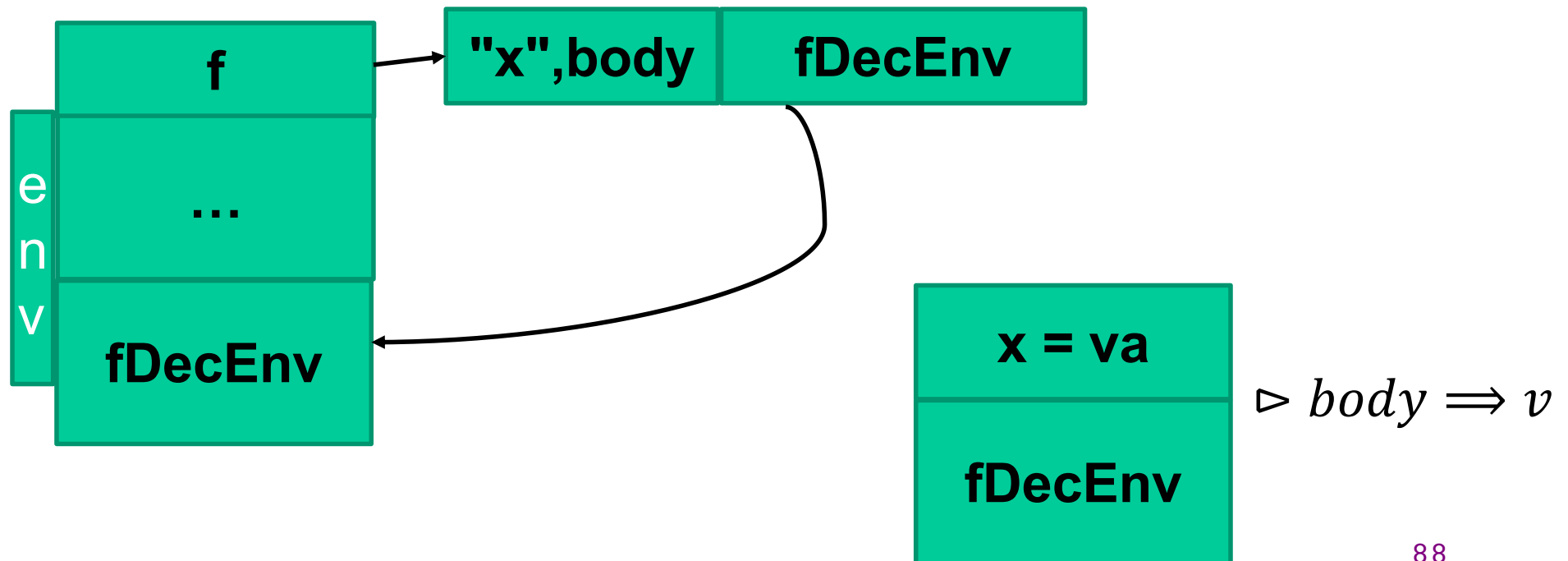
Restituisce la
Closure(x, e, s)

corpo del let

$\Sigma[f=Closure("x", e, \Sigma)]$

Applicazione di una funzione

$$\begin{array}{l}
 \Sigma \triangleright Den("f") \Rightarrow Closure("x", body, \Sigma_{fDecl}) \\
 \Sigma \triangleright arg \Rightarrow va \quad \Sigma_{fDecl}[x = va] \triangleright body \Rightarrow v \\
 \hline
 \Sigma \triangleright Apply(Den("f"), arg) \Rightarrow v
 \end{array}$$



Le regole dell'interprete (scoping statico)



let rec eval (e: exp) (s: evT env) : evT =

match e **with**

| ...

| **Fun**(i, a) -> **Closure**(i, a, s)

$\Sigma \triangleright \text{Fun}(x, e) \Rightarrow \text{Closure}("x", e, \Sigma)$

Si introduce la chiusura
Closure("x", e, Σ)

Le regole dell'interprete (scoping statico)

let rec eval (e: exp) (s: evT env) : evT =

match e **with**

| ...

| **Fun**(i, a) -> **Closure**(i, a, s)

| **Apply**(**Den**(f), eArg) ->

let **fclosure** = s f in

(**match** **fclosure** with

| **Closure**(arg, fbody, fDecEnv) ->

$$\frac{\begin{array}{l} \Sigma \triangleright \text{Den}("f") \Rightarrow \text{Closure}("x", \text{body}, \Sigma_{f\text{Decl}}) \\ \Sigma \triangleright \text{arg} \Rightarrow va \quad \Sigma_{f\text{Decl}}[x = va] \triangleright \text{body} \Rightarrow v \end{array}}{\Sigma \triangleright \text{Apply}(\text{Den}("f"), \text{arg}) \Rightarrow v}$$

Si recupera, dall'ambiente corrente (**s**), cioè del chiamante, il valore associato a **f**, ovvero la chiusura **Closure**

$$\Sigma \triangleright \text{Den}("f") \Rightarrow \text{fclosure}$$

$$\text{Closure}("x", \text{body}, \Sigma_{f\text{Decl}})$$

Le regole dell'interprete (scoping statico)



let rec eval (e: exp) (s: evT env) : evT =

match e **with**

| ...

| Fun(i, a) -> Closure(i, a, s)

| Apply(Den(f), eArg) ->

let fclosure = s f in

(match fclosure with

| Closure(arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

$$\frac{\begin{array}{l} \Sigma \triangleright \text{Den}("f") \Rightarrow \text{Closure}("x", \text{body}, \Sigma_{f\text{Decl}}) \\ \Sigma \triangleright \text{arg} \Rightarrow va \quad \Sigma_{f\text{Decl}}[x = va] \triangleright \text{body} \Rightarrow v \end{array}}{\Sigma \triangleright \text{Apply}(\text{Den}("f"), \text{arg}) \Rightarrow v}$$

Si valuta il parametro attuale
viene nell'ambiente corrente (**s**),
cioè del chiamante, ottenendo
aVal

$$\Sigma \triangleright \text{arg} \Rightarrow va$$

Le regole dell'interprete (scoping statico)



let rec eval (e: exp) (s: evT env) : evT =

match e **with**

| ...

| **Fun**(i, a) -> **Closure**(i, a, s)

| **Apply**(**Den**(f), eArg) ->

let fclosure = s f in

(match fclosure with

| **Closure**(arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let **aenv** = bind fDecEnv arg aVal in

$$\begin{array}{c} \Sigma \triangleright \text{Den}("f") \Rightarrow \text{Closure}("x", \text{body}, \Sigma_{f\text{Decl}}) \\ \Sigma \triangleright \text{arg} \Rightarrow va \quad \Sigma_{f\text{Decl}}[x = va] \triangleright \text{body} \Rightarrow v \\ \hline \Sigma \triangleright \text{Apply}(\text{Den}("f"), \text{arg}) \Rightarrow v \end{array}$$

L'ambiente al momento della dichiarazione ((**fDecEnv**)) viene esteso con l'associazione tra il parametro **arg** e il valore **aVal**: si ottiene **aenv**

$$\Sigma' = \Sigma_{f\text{Decl}}[x = va]$$

Le regole dell'interprete (scoping statico)



let rec eval (e: exp) (s: evT env) : evT =

match e **with**

| ...

| Fun(i, a) -> Closure(i, a, s)

| Apply(Den(f), eArg) ->

let fclosure = s f in

(match fclosure with

| Closure(arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let aenv = bind fDecEnv arg aVal in

eval fbody aenv

$$\frac{\begin{array}{l} \Sigma \triangleright Den("f") \Rightarrow Closure("x", body, \Sigma_{fDecl}) \\ \Sigma \triangleright arg \Rightarrow va \quad \Sigma_{fDecl}[x = va] \triangleright body \Rightarrow v \end{array}}{\Sigma \triangleright Apply(Den("f"), arg) \Rightarrow v}$$

La funzione valuta (**eval**) il
corpo della funzione (**fbody**)
in **aenv**
 $\Sigma_{fDecl}[x = va] \triangleright body \Rightarrow v$

Le regole dell'interprete (scoping statico)



let rec eval (e: exp) (s: evT env) : evT =

match e **with**

| ...

| Fun(i, a) -> Closure(i, a, s)

| Apply(Den(f), eArg) ->

let fclosure = s f in

(match fclosure with

| Closure(arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let aenv = bind fDecEnv arg aVal in

eval fbody aenv

| _ -> failwith("non functional value"))

| Apply(_,_) -> failwith("Application: not first order function") ;;

$$\frac{\Sigma \triangleright Den("f") \Rightarrow Closure("x", body, \Sigma_{fDecl}) \quad \Sigma \triangleright arg \Rightarrow va \quad \Sigma_{fDecl}[x = va] \triangleright body \Rightarrow v}{\Sigma \triangleright Apply(Den("f"), arg) \Rightarrow v}$$

Passi dell'interprete per l'applicazione

- Si recupera, dall'ambiente corrente (**s**), il valore associato a **f**, ovvero la chiusura **Closure**
- Si valuta il parametro attuale nell'ambiente corrente (**s**), cioè del chiamante, ottenendo il valore **aVal**
- L'ambiente al momento della dichiarazione(**fDecEnv**) viene esteso con l'associazione tra il parametro **arg** e il valore **aVal**: si ottiene **aenv**
- La funzione valuta (**eval**) il corpo della funzione **fbody** in **aenv**

SCOPING STATICO: *il corpo della funzione viene valutato nell'ambiente ottenuto legando il parametro formale al valore del parametro attuale nell'ambiente nel quale era stata dichiarata l'astrazione*

Semantica operativa insieme a quella eseguibile

$$\frac{\begin{array}{l} \Sigma \triangleright Den("f") \Rightarrow Closure("x", body, \Sigma_{fDecl}) \\ \Sigma \triangleright arg \Rightarrow va \quad \Sigma_{fDecl}[x = va] \triangleright body \Rightarrow v \end{array}}{\Sigma \triangleright Apply(Den("f"), arg) \Rightarrow v}$$

```
let rec eval (e: exp) (s: evT env) : evT =  
  match e with  
  | ...  
  | Fun(i, a) -> Closure(i, a, s)  
  | Apply(Den(f), eArg) ->  
    let fclosure = s f in  
    (match fclosure with  
    | Closure(arg, fbody, fDecEnv) ->  
      let aVal = eval eArg s in  
      let aenv = bind fDecEnv arg aVal in  
      eval fbody aenv  
    | _ -> failwith("non functional value"))  
  | Apply(_,_) -> failwith("Application: not first order function") ;;
```

REPL

```
let x = 5 in
  let f = fun z -> z + x in f 1
```

```
# let e = Let ("x", CstInt 5,
  Let ("f", Fun ("z", Sum (Den "z", Den "x")), Apply (Den "f", CstInt 1))));;
val e1 : exp =
  Let ("x", CstInt 5,
    Let ("f", Fun ("z", Sum (Den "z", Den "x")), Apply (Den "f", CstInt 1)))
# eval e1 emptyEnv ;;
- : evT = Int 6
```

REPL

```
let x = 5 in  
  let f = fun z -> z + x in f 1
```

```
# let e = Let ("x", CstInt 5,  
  Let ("f", Fun ("z", Sum (Den "z", Den "x")), Apply (Den "f", CstInt 1))));;  
val e1 : exp =  
  Let ("x", CstInt 5,  
    Let ("f", Fun ("z", Sum (Den "z", Den "x")), Apply (Den "f", CstInt 1)))  
# eval e1 emptyEnv ;;  
- : evT = Int 6
```

Nota Bene: se ci fosse un “*let x = 10 in*” prima di *Apply*, il risultato non cambierebbe, a causa dello *scoping statico*: nella chiusura rimane l’ambiente di dichiarazione, in questo caso con l’associazione di *x* a 5

Scoping Dinamico

Scoping dinamico: dobbiamo modificare **evT**

```
type evT = | Int of int | Bool of bool | Unbound  
          | Funval of efun
```

e anche

```
efun = ide * exp
```

- La definizione di **efun** mostra che l'astrazione funzionale, oltre al parametro formale, contiene **solo il corpo della funzione dichiarata** (**non** serve catturare l'ambiente al momento della definizione)
- Il corpo della funzione verrà quindi valutato nell'**ambiente nel quale avviene l'applicazione**, legando i parametri formali ai valori dei parametri attuali

Astrazione e applicazione di funzione: scoping dinamico

SINTASSI ASTRATTA

Funval of efun

efun = ide * exp

Apply of Den("f") * exp

- Identificatori (**parametri formali**) nel costrutto di astrazione

Funval of ide * exp

- Espressioni (**parameri attuali**) nel costrutto di applicazione

Apply of Den("f") * exp

Astrazione e applicazione di funzione: scoping dinamico

Astrazione funzionale

contiene **solo** il parametro formale e il corpo della funzione dichiarata (*e*)

$$\Sigma \triangleright Fun("x", e) \Rightarrow Funval("x", e)$$

Applicazione

Il corpo della funzione (*e*) viene valutato nell'**ambiente nel quale avviene l'applicazione** (Σ), dopo aver legato i parametri formali (*x*) ai valori dei parametri attuali (**arg**)

$$\begin{array}{l} \Sigma \triangleright Den("f") \Rightarrow Funval("x", e) \\ \Sigma \triangleright arg \Rightarrow va \quad \Sigma[x = va] \triangleright e \Rightarrow v \\ \hline \Sigma \triangleright Apply(Den("f"), arg) \Rightarrow v \end{array}$$

Scoping dinamico: interprete

let rec eval (e: exp) (s: evT env) : evT =
 match e **with**

| ...

| **Fun**(arg, ebody) -> **Funval**(arg,ebody)

$\Sigma \triangleright Den("f") \Rightarrow Funval("x", e)$

Il valore associato alla
funzione contiene **solo** il
parametro formale e il corpo
della funzione dichiarata (**e**)

Scoping dinamico: interprete

let rec eval (e: exp) (s: evT env) : evT =

match e **with**

| ...

| Fun(arg, ebody) -> Funval(arg, ebody)

| Apply(Den(f), eArg) ->

let fval = s f in

(match fval with

| Funval(arg, fbody) ->

$$\frac{\Sigma \triangleright Den("f") \Rightarrow Funval("x", e) \quad \Sigma \triangleright arg \Rightarrow va \quad \Sigma[x = va] \triangleright e \Rightarrow v}{\Sigma \triangleright Apply(Den("f"), arg) \Rightarrow v}$$

Si recupera, dall'ambiente corrente (s),
il valore associato a **f**

$$\Sigma \triangleright Den("f") \Rightarrow Funval("x", fbody)$$

Scoping dinamico: interprete

let rec eval (e: exp) (s: evT env) : evT =

match e **with**

| ...

| Fun(arg, ebody) -> Funval(arg, ebody)

| Apply(Den(f), eArg) ->

let fval = s f in

(match fval with

| Funval(arg, fbody) ->

let aVal = eval eArg s in

$$\frac{\begin{array}{l} \Sigma \triangleright \text{Den}("f") \Rightarrow \text{Funval}("x", e) \\ \Sigma \triangleright \text{arg} \Rightarrow va \quad \Sigma[x = va] \triangleright e \Rightarrow v \end{array}}{\Sigma \triangleright \text{Apply}(\text{Den}("f"), \text{arg}) \Rightarrow v}$$

Si valuta il parametro attuale nell'ambiente corrente (**s**), cioè del chiamante, ottenendo il valore **aVal**

$$\Sigma \triangleright \text{arg} \Rightarrow va$$

Scoping dinamico: interprete

let rec eval (e: exp) (s: evT env) : evT =

match e **with**

| ...

| Fun(arg, ebody) -> Funval(arg, ebody)

| Apply(Den(f), eArg) ->

let fval = s f in

(match fval with

| Funval(arg, fbody) ->

let aVal = eval eArg s in

let aenv = bind s arg aVal in

$$\frac{\begin{array}{l} \Sigma \triangleright \text{Den}("f") \Rightarrow \text{Funval}("x", e) \\ \Sigma \triangleright \text{arg} \Rightarrow va \quad \Sigma[x = va] \triangleright e \Rightarrow v \end{array}}{\Sigma \triangleright \text{Apply}(\text{Den}("f"), \text{arg}) \Rightarrow v}$$

L'ambiente corrente viene esteso con l'associazione tra il parametro **arg** e il valore **aVal**: si ottiene **aenv**

$$\Sigma' = \Sigma[x = va]$$

Scoping dinamico: interprete

let rec eval (e: exp) (s: evT env) : evT =

match e **with**

| ...

| Fun(arg, ebody) -> Funval(arg, ebody)

| Apply(Den(f), eArg) ->

let fval = s f in

(match fval with

| Funval(arg, fbody) ->

let aVal = eval eArg s in

let aenv = bind s arg aVal in

eval fbody aenv

$$\frac{\begin{array}{l} \Sigma \triangleright \text{Den}("f") \Rightarrow \text{Funval}("x", e) \\ \Sigma \triangleright \text{arg} \Rightarrow va \quad \Sigma[x = va] \triangleright e \Rightarrow v \end{array}}{\Sigma \triangleright \text{Apply}(\text{Den}("f"), \text{arg}) \Rightarrow v}$$

La funzione valuta (**eval**) il corpo della funzione (**fbody**) in **aenv**

$$\Sigma[x = va] \triangleright e \Rightarrow v$$

Scoping dinamico: interprete

let rec eval (e: exp) (s: evT env) : evT =

match e **with**

| ...

| Fun(arg, ebody) -> Funval(arg,ebody)

| Apply(Den(f), eArg) ->

let fval = s f in

(match fval with

| Funval(arg, fbody) ->

let aVal = eval eArg s in

let aenv = bind s arg aVal in

eval fbody aenv

failwith("non functional value"))

| Apply(_,_) -> failwith("Application: not first order function") ;;

$$\frac{\begin{array}{l} \Sigma \triangleright Den("f") \Rightarrow Funval("x", e) \\ \Sigma \triangleright arg \Rightarrow va \quad \Sigma[x = va] \triangleright e \Rightarrow v \end{array}}{\Sigma \triangleright Apply(Den("f"), arg) \Rightarrow v}$$

Passi dell'interprete per l'applicazione

- Si recupera, dall'ambiente corrente (**s**), il valore associato a **f**
- Si valuta il parametro attuale nell'ambiente corrente (**s**), cioè del chiamante, ottenendo il valore **aVal**
- L'ambiente corrente(**s**) viene esteso con l'associazione tra il parametro **arg** e il valore **aVal**: si ottiene **aenv**
- La funzione valuta (**eval**) il corpo della funzione **fbody** in **aenv**

SCOPING DINAMICO: *il corpo della funzione viene valutato nell'ambiente ottenuto legando il parametro formale al valore del parametro attuale nell'ambiente corrente, cioè dove viene effettuata la chiamata*

Ricapitolando: regole di scoping

Scoping statico (lessicale): l'ambiente non locale della funzione è quello esistente al momento in cui viene dichiarata l'astrazione

```
Closure(arg, fbody, fDecEnv) ->  
    let aVal = eval eArg s in  
    let aenv = bind fDecEnv arg aVal in  
    eval fbody aenv
```

Scoping dinamico: l'ambiente non locale della funzione è quello esistente al momento nel quale avviene l'applicazione

```
Funval(arg, fbody) ->  
    let aVal = eval eArg s in  
    let aenv = bind s arg aVal in  
    eval fbody aenv
```

Nel **linguaggio didattico** adottiamo lo **scoping statico**

Definizioni ricorsive

Funzioni ricorsive

- Come è fatta una definizione di funzione ricorsiva?
- È una espressione $\text{Let}(f, e1, e2)$ in cui:
 - f è il nome della funzione (ricorsiva)
 - $e1$ è un'astrazione $\text{Fun}(i, a)$, nel cui corpo occorre un'applicazione di Den " f "

Esempio

```
Let("fact",  
    Fun("x", Ifthenelse(Eq(Den "x", CstInt(0)),  
                          Eint CstInt(1),  
                          Prod(Den "x",  
                              Appl(Den "fact",  
                                   [Diff(Den  
"x", CstInt(1))])))),  
    Appl(Den "fact", [CstInt(4)]))
```

In OCaml

```
let rec fact x =  
  if (x == 0) then 1 else (x * fact(x-1)) in fact(4)
```

Il nostro interprete attuale non
funziona con funzioni ricorsive

Guardiamo la semantica

```
| Let(i, e1, e2) -> eval e2 (bind s i (eval e1 s))
| Fun(i, a) -> Closure(i, a, s)
| Apply(Den(f), eArg) ->
    let fclosure = s f in
    (match fclosure with
    | Closure(arg, fbody, fDecEnv) ->
        let aVal = eval eArg s in
        let aenv = bind fDecEnv arg aVal in
        eval fbody aenv
    | _ -> failwith("non functional value"))
| Apply(_,_) -> failwith("Application: not first order function")
```

Il corpo **a** (che include **Den("fact")**) è valutato in un ambiente (**aenv**) che estende **fDecEnv = s** con un'associazione per il parametro formale **x**

Tuttavia, **s non** contiene legami per il nome **"fact"**, e quindi **Den("fact")** restituisce **Unbound!**

Morale

- Per permettere la **ricorsione**, bisogna che il corpo della funzione venga valutato in un **ambiente nel quale è già stata inserita l'associazione tra il nome e la funzione**
- Abbiamo bisogno quindi di
 - un diverso costrutto per “dichiarare” funzioni ricorsive (come il **let rec** di OCaml)
 - oppure di un diverso costrutto di astrazione per le funzioni ricorsive

Problema generale

- Come costruiamo la **chiusura** per la gestione della ricorsione?
- Il punto importante è che **l'ambiente della chiusura deve contenere un binding per la gestione della ricorsione**

Ricorsione

La **ricorsione** è gestita attraverso:

- un modo per **dichiarare una funzione ricorsiva**
- una **chiusura ricorsiva**, chiusura speciale che permette alla funzione di fare riferimento a sé stessa

Letrec

Estendiamo la sintassi astratta del linguaggio didattico con un opportuno costruttore

```
type exp =
```

```
| ...
```

```
| Letrec of ide * ide * exp * exp
```

```
Letrec("f", "x", fbody, letbody)
```

- **"f"** è il nome della funzione
- **"x"** parametro formale
- **fbody** corpo della funzione
- **letbody** corpo del let

Il solito fattoriale

```
Letrec("fact", "n",  
  Ifthenelse(Eq(Den("n"), CstInt(0),  
    CstInt(1)),  
    Times(Den("n"), Apply(Den("fact"), Sub(Den("n"), CstInt(1))))),  
  Apply(Den("fact"), CstInt(3)))
```

Valori esprimibili evT

È necessario estendere i valori esprimibili (evT) per avere le **astrazioni funzionali ricorsive** (**RecClosure**)

```
type evT = | Int of int | Bool of bool  
          | Unbound | Closure of ide * exp * evT env  
          | RecClosure of ide * ide * exp * evT env
```



RecFunVal

`RecClosure of ide * ide * exp * evT env`

```
RecClosure (funName,  
            param,  
            funBody,  
            staticEnvironment)
```

Interprete: dichiarazione funzione ricorsiva

```
let rec eval (e: exp) (s: evT env) : evT =  
  match e with  
  | ...  
  | Letrec(f, i, fBody, letBody) ->  
    let benv =  
      bind s f (RecClosure(f, i, fBody, s))
```

... Si associa il nome della funzione ricorsiva a una chiusura ricorsiva (**RecClosure**) che contiene: il nome della funzione stessa (**f**), i parametri (**i**), il corpo della funzione (**fBody**) e l'ambiente corrente, cioè di dichiarazione (**s**)

Si estende l'ambiente corrente (**s**) con il binding che associa **f** alla chiusura ricorsiva (**RecClosure(f, i, fBody, s)**), ottenendo il nuovo ambiente **benv**

Questo permette a **f** di essere visibile in **letBody**, consentendo di fare riferimento a sé stessa in modo ricorsivo

Interprete: dichiarazione funzione ricorsiva

```
let rec eval (e: exp) (s: evT env) : evT =  
  match e with  
  | ...  
  | Letrec(f, i, fBody, letBody) ->  
    let benv =  
      bind s f (RecClosure(f, i, fBody, s))  
    in eval letBody benv
```

...

L'interprete valuta (**eval**) il corpo del let (**letBody**) nell'ambiente esteso **benv**, permettendo di chiamare la funzione ricorsivamente all'interno di **letBody**

Passi dell'interprete per la ricorsione

La funzione di valutazione:

- associa il **nome della funzione ricorsiva** a una **chiusura ricorsiva** (**RecClosure**) che contiene: il **nome della funzione stessa (f)**, i parametri (**i**), il corpo della funzione (**fBody**) e l'ambiente corrente, cioè di dichiarazione (**s**)
- estende l'ambiente corrente (**s**) con il binding che associa **f** alla chiusura ricorsiva (**RecClosure(f, i, fBody, s)**), ottenendo il nuovo ambiente **benv**. Questo permette a **f** di essere visibile in **letBody**, consentendo di fare riferimento a sé stessa in modo ricorsivo
- valuta il corpo del let (**letBody**), cioè l'espressione da valutare con la funzione ricorsiva, nell'ambiente aggiornato

Interprete: applicazione funzione ricorsiva

...

| Apply(Den f, eArg) ->

let **fclosure** = s f in

match fclosure with

| Closure(arg, fbody, fDecEnv) -> ...

| **RecClosure**(f, arg, fbody, fDecEnv) ->

Il valore della chiusura ricorsiva **RecClosure** viene recuperato dall'ambiente corrente

Interprete: applicazione funzione ricorsiva

...

| Apply(Den f, eArg) ->

let fclosure = s f in

match fclosure with

| Closure(arg, fbody, fDecEnv) -> ...

| RecClosure(f, arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

Si valuta il parametro attuale (**eArg**) nell'ambiente corrente (**s**), cioè del chiamante, ottenendo il valore **aVal**

Interprete: applicazione funzione ricorsiva

...

| **Apply**(Den f, eArg) ->

let fclosure = s f in

match fclosure with

| **Closure**(arg, fbody, fDecEnv)

| **RecClosure**(f, arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let **rEnv** = bind fDecEnv f fclosure in

Affinché la funzione possa chiamare sé stessa ricorsivamente, l'ambiente statico **fDecEnv**, memorizzato nella chiusura, viene esteso con il legame tra il nome della funzione (**f**) e **fclosure** (che è la sua chiusura ricorsiva **RecClosure**), ottenendo l'ambiente **rEnv**, che permette alla funzione di fare riferimento a sé stessa

Interprete: applicazione funzione ricorsiva

...

| Apply(Den f, eArg) ->

let fclosure = s f in

match fclosure with

| Closure(arg, fbody, fDecEnv) -> ...

| RecClosure(f, arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let rEnv = bind fDecEnv f fclosure in

let aEnv = bind rEnv arg aVal in

L'ambiente effettivo di esecuzione **aEnv** viene infine ottenuto estendendo l'ambiente **rEnv** con l'associazione tra il parametro **arg** e il valore **aVal** (passaggio del parametro)

Interprete: applicazione funzione ricorsiva

...

| Apply(Den f, eArg) ->

let fclosure = s f in

match fclosure with

| Closure(arg, fbody, fDecEnv) -> ...

| RecClosure(f, arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let rEnv = bind fDecEnv f fclosure in

let aEnv = bind rEnv arg aVal in

eval(fbody, aEnv)

La funzione valuta (**eval**) **fbody** in **aenv**, che contiene sia l'accesso ricorsivo a **f** che il valore dell'argomento **arg**

Interprete: applicazione funzione ricorsiva

...

| Apply(Den f, eArg) ->

let fclosure = s f in

match fclosure with

| Closure(arg, fbody, fDecEnv) -> ...

| RecClosure(f, arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let rEnv = bind fDecEnv f fclosure in

let aEnv = bind rEnv arg aVal in

eval(fbody, aEnv)

| _ -> failwith("non functional value")

| Apply(_,_) -> failwith("not function")

Gestione eventuali errori

Interprete: applicazione funzione ricorsiva

...

```
| Apply(Den f, eArg) ->  
  let fclosure = s f in  
  match fclosure with  
  | Closure(arg, fbody, fDecEnv) -> ...  
  | RecClosure(f, arg, fbody, fDecEnv) ->  
    let aVal = eval eArg s in  
    let rEnv = bind fDecEnv f fclosure in  
    let aEnv = bind rEnv arg aVal in  
    eval(fbody, aEnv)  
  | _ -> failwith("non functional value")  
| Apply(_,_) -> failwith("not function")
```


Passi dell'interprete per l'applicazione

- Il valore della chiusura ricorsiva **RecClosure** viene recuperato dall'ambiente corrente (**s**)
- Il parametro attuale viene valutato nell'ambiente corrente (**s**), cioè del chiamante, ottenendo il valore **aVal**
- Affinché la funzione possa chiamare sé stessa ricorsivamente, l'ambiente statico **fDecEnv**, memorizzato nella chiusura, viene esteso con il legame tra il nome della funzione (**f**) e **fclosure** (che è la sua chiusura ricorsiva, cioè **RecClosure**), ottenendo l'ambiente **rEnv**, che permette alla funzione di fare riferimento a sé stessa
- L'ambiente effettivo di esecuzione **aEnv** viene infine ottenuto estendendo l'ambiente **rEnv** con l'associazione tra il parametro **arg** e il valore **aVal** (passaggio del parametro)
- La funzione valuta (**eval**) **fbody** in **aenv**, che contiene sia l'accesso ricorsivo a **f** che il valore dell'argomento **arg**

REPL

```
# let myRP =
  Letrec("fact", "n",
    Ifthenelse(Eq(Den("n"), EInt(0)),
      EInt(1),
      Prod(Den("n"),
        Apply(Den("fact"),
          Sub(Den("n"), CstInt(1))))),
    Apply(Den("fact"), EInt(3))) ;;

val myRP : exp = ...

# eval myRP emptyEnv;;
- : eval = Int 6
```

Higher Order Functions

- Estendiamo la sintassi del linguaggio didattico per avere al possibilità di trattare **funzioni come valori di prima classe**
- Questo significa ammettere la possibilità che una funzione possa accettare altre funzioni come argomenti o restituire funzioni come risultati

Higher Order Function: sintassi astratta

Estendiamo la sintassi astratta del linguaggio

exp ::= ... | **Apply** of **exp** * **exp**

L'interprete valuta l'applicazione funzionale

Apply(eF, eArg) nel modo seguente:

- valuta l'espressione della funzione (**eF**) per ottenere un valore funzionale (una **chiusura**, semplice o ricorsiva)
- valuta l'argomento (**eArg**) nell'ambiente corrente
- stende l'ambiente della chiusura con il binding del parametro formale al valore dell'argomento
- valuta **il corpo della funzione (estratto dalla chiusura)** nell'ambiente esteso

Interprete

| Apply(**eF**, eArg) ->
let **fclosure** = eval **eF** s in
(match fclosure with

Valuta (**eval**) l'espressione della funzione (**eF**)
per ottenere **fclosure**, cioè un valore funzionale
(una **chiusura**, semplice o ricorsiva)

Interprete

```
| Apply(eF, eArg) ->  
  let fclosure = eval eF s in  
    (match fclosure with  
    | Closure(arg, fbody, fDecEnv) ->
```

CASO 1: chiusura semplice

Interprete

| Apply(**eF**, eArg) ->

let fclosure = **eval eF s** in

(match fclosure with

| Closure(arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

Si valuta il parametro attuale (**eArg**) nell'ambiente corrente, cioè del chiamante, ottenendo il valore **aVal**

Interprete

| Apply(**eF**, eArg) ->

let fclosure = **eval eF s** in

(match fclosure with

| **Closure**(arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let **aenv** = bind fDecEnv arg aVal in

L'ambiente al momento della Dichiarazione della funzione (**fDecEnv**) viene esteso con l'associazione tra il parametro **arg** e il valore **aVal**: si ottiene **aenv**

Interprete

```
| Apply(eF, eArg) ->  
  let fclosure = eval eF s in  
    (match fclosure with  
    | Closure(arg, fbody, fDecEnv) ->  
      let aVal = eval eArg s in  
        let aenv = bind fDecEnv arg aVal in  
          eval fbody aenv)
```

La funzione valuta (**eval**) il
corpo della funzione (**fbody**)
in **aenv**

Interprete

| Apply(**eF**, eArg) ->

let fclosure = **eval eF s** in

(match fclosure with

| Closure(arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let aenv = bind fDecEnv arg aVal in

eval fbody aenv

CASO 2: chiusura ricorsiva

RecClosure(f, arg, fbody, fDecEnv) ->

Interprete

| Apply(**eF**, eArg) ->

let fclosure = **eval eF s** in

(match fclosure with

| Closure(arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let aenv = bind fDecEnv arg aVal in

eval fbody aenv

| RecClosure(f, arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

Si valuta il parametro attuale (**eArg**) nell'ambiente corrente, cioè del chiamante, ottenendo il valore **aVal**

Interprete

| **Apply**(**eF**, **eArg**) ->

let **fclosure** = **eval eF s** in

(match **fclosure** with

| **Closure**(**arg**, **fbody**, **fDecEnv**)

let **aVal** = **eval eArg s** in

let **aenv** = **bind fDecEnv**

eval fbody aenv

| **RecClosure**(**f**, **arg**, **fbody**, **fDecEnv**) ->

let **aVal** = **eval eArg s** in

let **rEnv** = **bind fDecEnv f fclosure** in

Affinché la funzione possa chiamare sé stessa ricorsivamente, l'ambiente statico **fDecEnv**, memorizzato nella chiusura, viene esteso con il legame tra il nome della funzione (**f**) e **fclosure** (che è la sua chiusura ricorsiva **RecClosure**), ottenendo l'ambiente **rEnv**, che permette alla funzione di fare riferimento a sé stessa

Interprete

| Apply(**eF**, eArg) ->

let fclosure = **eval eF s** in

(match fclosure with

| Closure(arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let aenv = bind fDecEnv arg aVal in

eval fbody aenv

| RecClosure(f, arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let rEnv = bind fDecEnv f fclosure in

let **aenv** = bind rEnv arg aVal in

L'ambiente effettivo di esecuzione **aEnv** viene quindi ottenuto estendendo l'ambiente **rEnv** con l'associazione tra il parametro **arg** e il valore **aVal** (passaggio del parametro)

Interprete

| Apply(**eF**, eArg) ->

let fclosure = **eval eF s** in

(match fclosure with

| Closure(arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let aenv = bind fDecEnv arg aVal in

eval fbody aenv

| RecClosure(f, arg, fbody, fDecEnv) ->

let aVal = eval eArg s in

let rEnv = bind fDecEnv f fclosure in

let aenv = bind rEnv arg aVal in

eval fbody aenv

La funzione valuta (**eval**) il
corpo della funzione (**fbody**)
in **aenv**

Interprete

```
| Apply(eF, eArg) ->
  let fclosure = eval eF s in
  (match fclosure with
  | Closure(arg, fbody, fDecEnv) ->
    let aVal = eval eArg s in
    let aenv = bind fDecEnv arg aVal in
    eval fbody aenv
  | RecClosure(f, arg, fbody, fDecEnv) ->
    let aVal = eval eArg s in
    let rEnv = bind fDecEnv f fclosure in
    let aenv = bind rEnv arg aVal in
    eval fbody aenv
  | _ -> failwith("non functional value")) ;;
```



REPL

```
# let apply_twice =
Fun("f", Fun("x", Apply(Den("f"),      Apply(Den("f"), Den("x")))));;

# let increment = Fun("x", Sum(Den("x"), CstInt (1)));;
val increment : exp = Fun ("x", Sum (Den "x", CstInt (1))

# let apply_twice_closure = eval apply_twice emptyenv;;
# let increment_closure = eval increment emptyenv;;

# let env_with_functions = bind (bind emptyenv "apply_twice"
apply_twice_closure) "increment" increment_closure;;

# let expression = Apply(Apply(Den("apply_twice"), Den("increment")),
CstInt (5));;

# let result = eval expression env_with_functions;;
  val result : evT = Int 7
```