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

Binding statico & dinamico

- Il termine **static binding** significa che i legami nome-entità sono definiti *prima* di mandare il programma in esecuzione
- Il termine **dynamic binding** significa che 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';
   }
}
```

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**
- **Blocco associato a una funzione**: corpo della funzione con le dichiarazioni dei parametri formali
- **Blocco in-line**: meccanismo per raggruppare comandi (es. corpo di un ciclo)

Blocchi

```
A: {  int aVar;
      aVar = 2;
      B: {  char aVar;
                aVar = 'a';
         }
   }
```

blocco in-line

apri blocco A;
apri blocco B;
chiudi blocco B;
chiudi blocco A;

politica di accesso ai blocchi: LIFO

I cambiamenti dell'ambiente avvengono all'entrata e all'uscita dai blocchi (anche annidati)

Tipi di ambiente

- ***Ambiente locale***: **l'insieme delle associazioni dichiarate localmente al blocco, compreso 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 usabili da tutte le componenti che costituiscono il programma**

JavaScript: scope (let vs var)

```
{
  let x = 2;
}
// x non visible qui


{
  var x = 2;
}
// x visible qui
```

JavaScript Ambiente locale

```
// carName non visibile

function myFunction() {
  let carName = "Volvo";
  // ambiente locale
  // contiene carName
}

// carName non visibile
```

Ambiente locale: JS Function Scope

```
function myFunction() {
  var carName = ”Ferrari";    // Function Scope
}
function myFunction() {
  let carName = ”Ferrari";    // Function Scope
}

function myFunction() {
  const carName = ”Ferrari";  // Function Scope
}
```

Ambiente globale: Javascript

```
let carName = ”Ferrari";
// si può riferire  carName

function myFunction() {
// si può riferire carName
}
```

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);
          }
      } } }
```

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;
```

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* di un binding definisce quella parte del programma nella quale il binding è attivo
  - **scope statico o lessicale**: è determinato dalla struttura sintattica del programma
  - **scope dinamico**: è determinato dalla struttura a tempo di esecuzione
- 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);
```

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);
```

Assumendo scoping dinamico questo codice stampa…. **2 3 2**

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

Operazioni su ambienti

- **Naming**: creazione di associazione fra nome e oggetto denotato (dichiarazione locale al blocco o parametro)
- **Referencing**: riferimento a un oggetto denotato mediante il suo nome (uso del nome per accedere all'oggetto denotato)
- **Disattivazione** di associazione fra nome e oggetto denotato (la nuova associazione per un nome maschera la vecchia associazione, che rimane disattivata fino all'uscita dal blocco)
- **Riattivazione** di associazione fra nome e oggetto denotato (una vecchia associazione che era mascherata è riattivata all'uscita da un blocco)
- **Unnaming**: distruzione di associazione fra nome e oggetto denotato (esempio: ambiente locale all'uscita di un blocco)
- **Il tempo di vita degli oggetti denotati non è necessariamente uguale al tempo di vita di un'associazione (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:

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

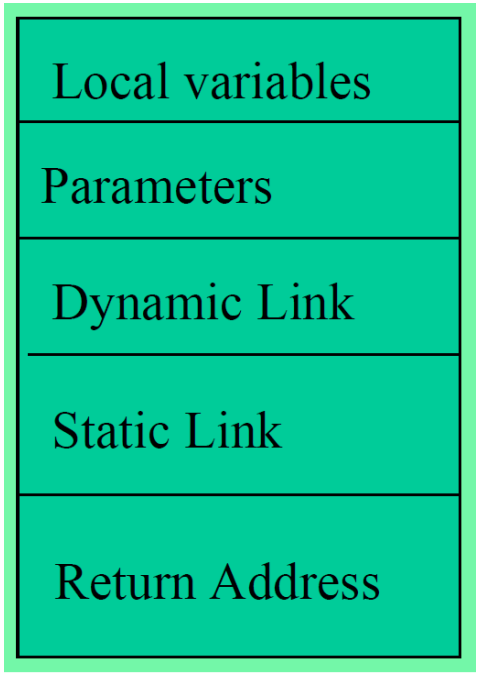
Variabili locali del blocco (incluso il risultato della funzione)

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

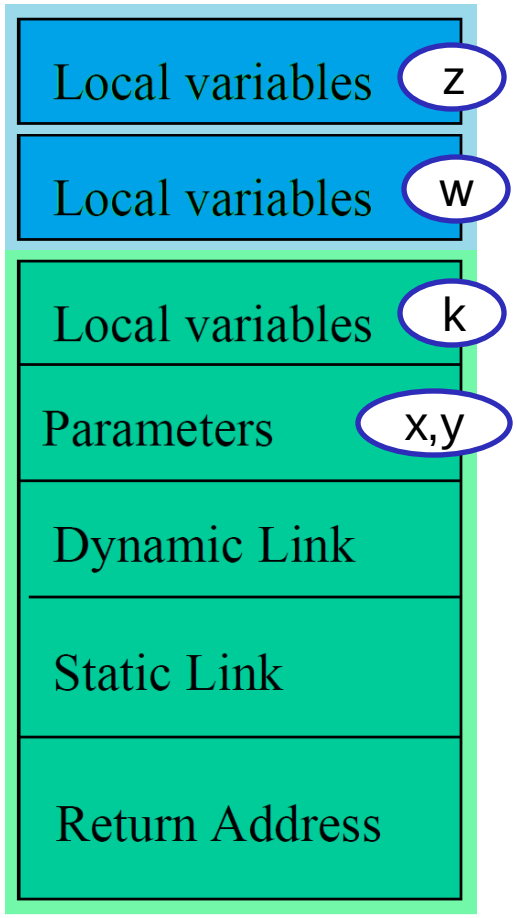


Lo stack

Riguardo i blocchi non associati a funzioni

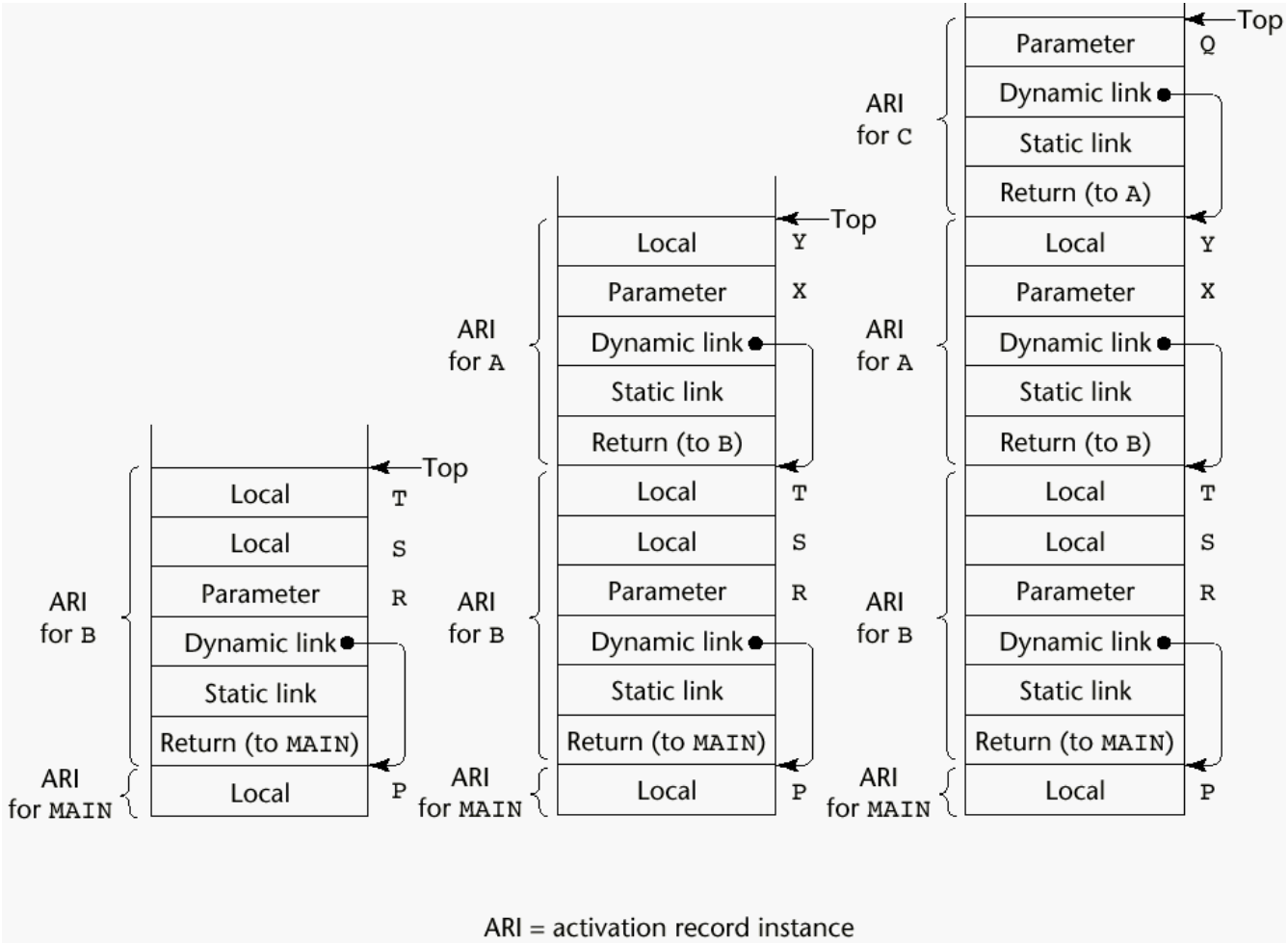
- La struttura è più semplice (solo local variables)
- Si impilano sul record di attivazione della funzione che li contiene
- Li ignoriamo per il momento

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



Esempio di esecuzione in C

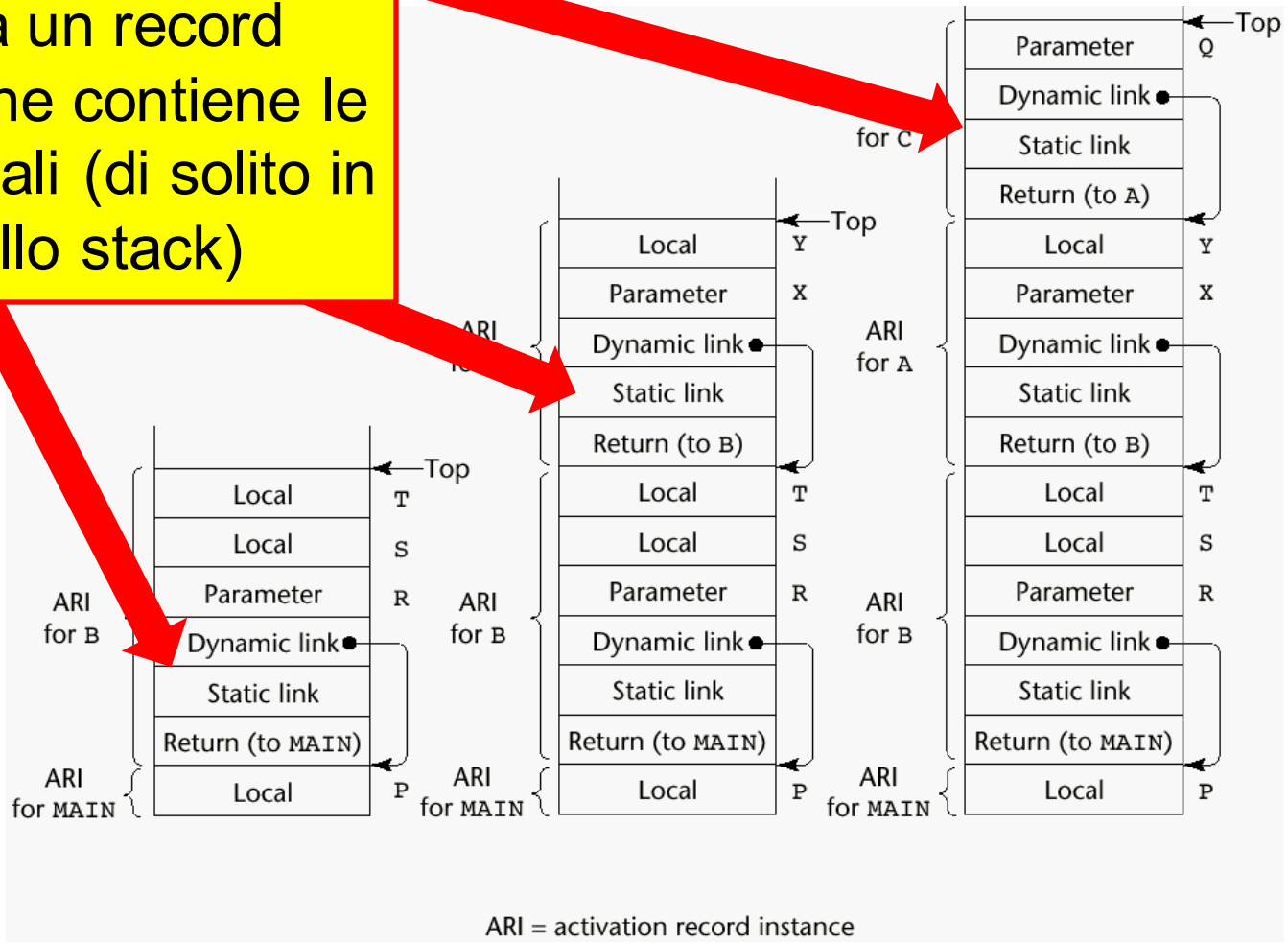
```
void A(int x) {
   int y;
   C(y);
}
void B(float r) {
   int s, t;
   A(s);
}
void C(int q) {
   ...
}
void main() {
   float p;
   B(p);
}
```



E ione in C

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)

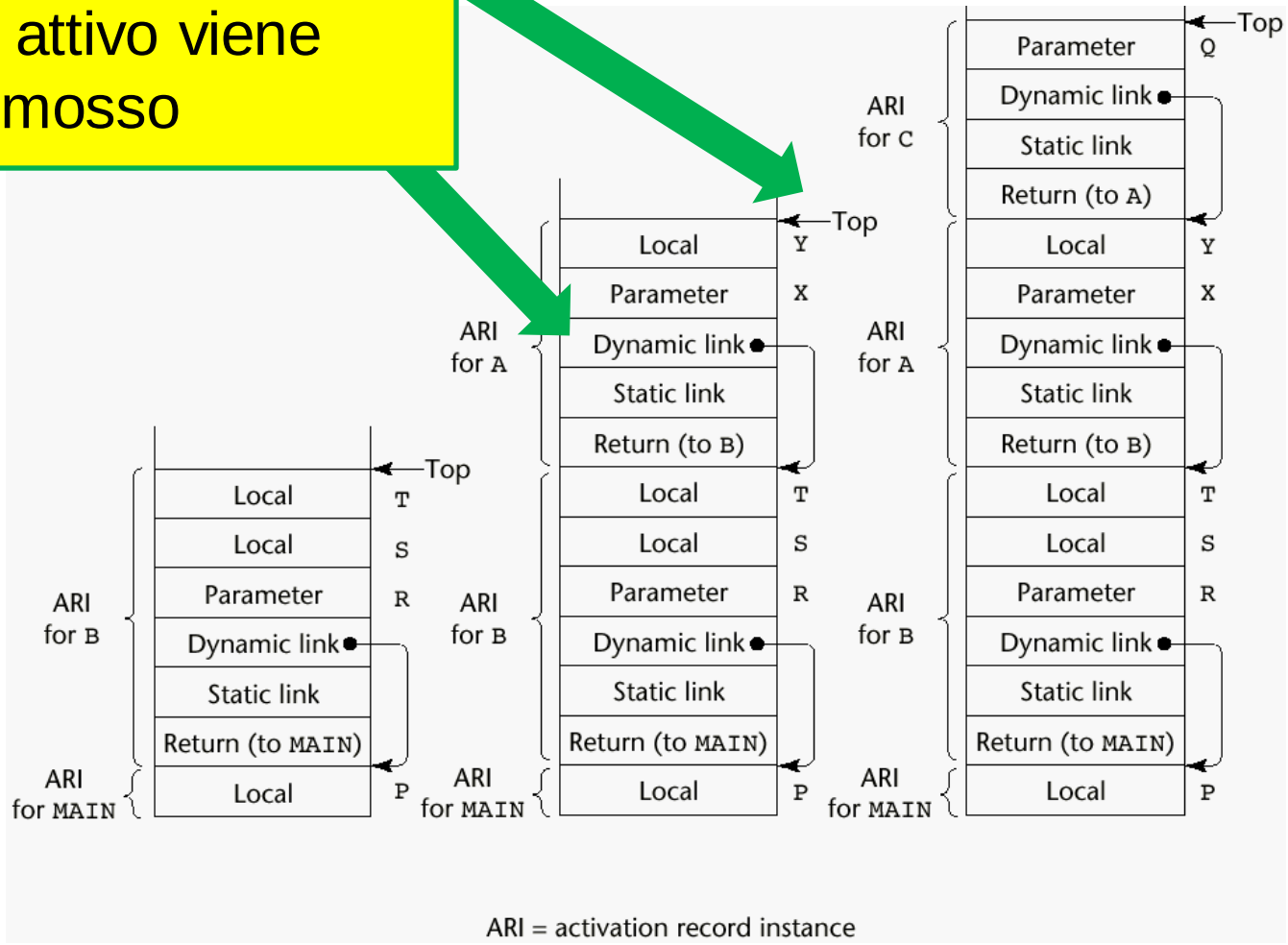
```
void A
   int
   C(y)
}
void B
   int s, t;
   A(s);
}
void C(int q) {
   ...
}
void main() {
   float p;
   B(p);
}
```



Es ne in C

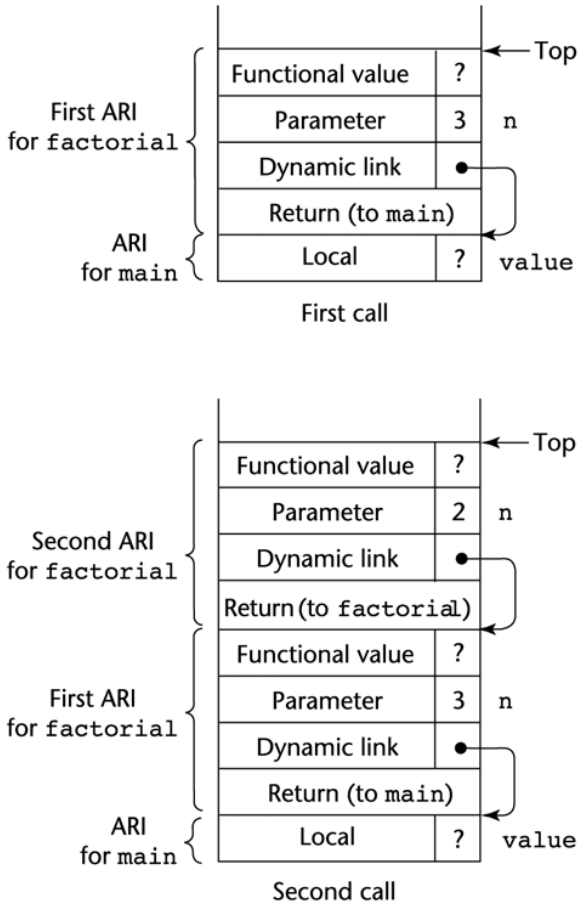
Il Dynamic Link serve a ripristinare il puntatore Top al termine dell'esecuzione della funzione, quando il record attivo viene rimosso

```
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);
}
```

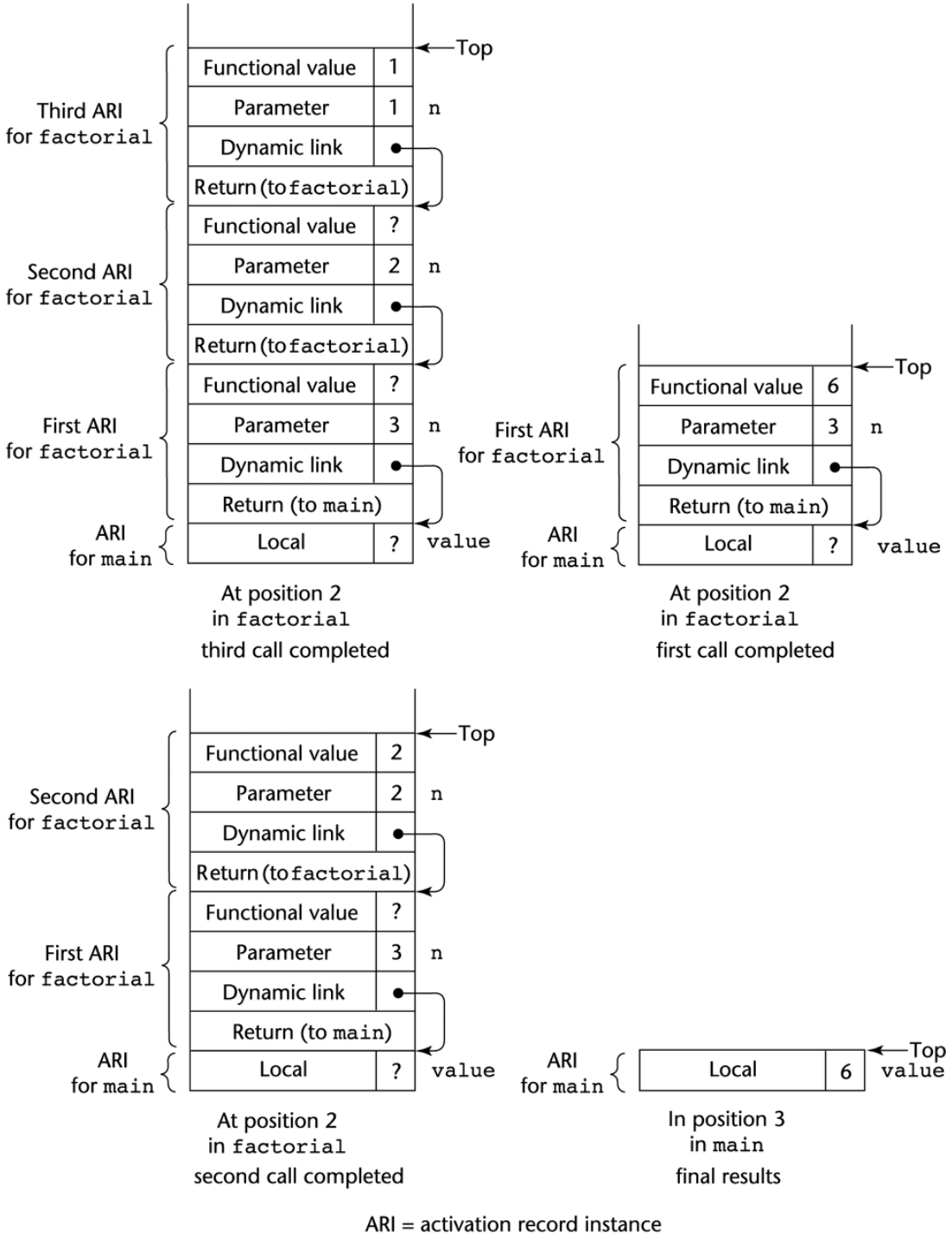


ARI = activation record instance

"Functional value" è il risultato della funzione

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);
}
```

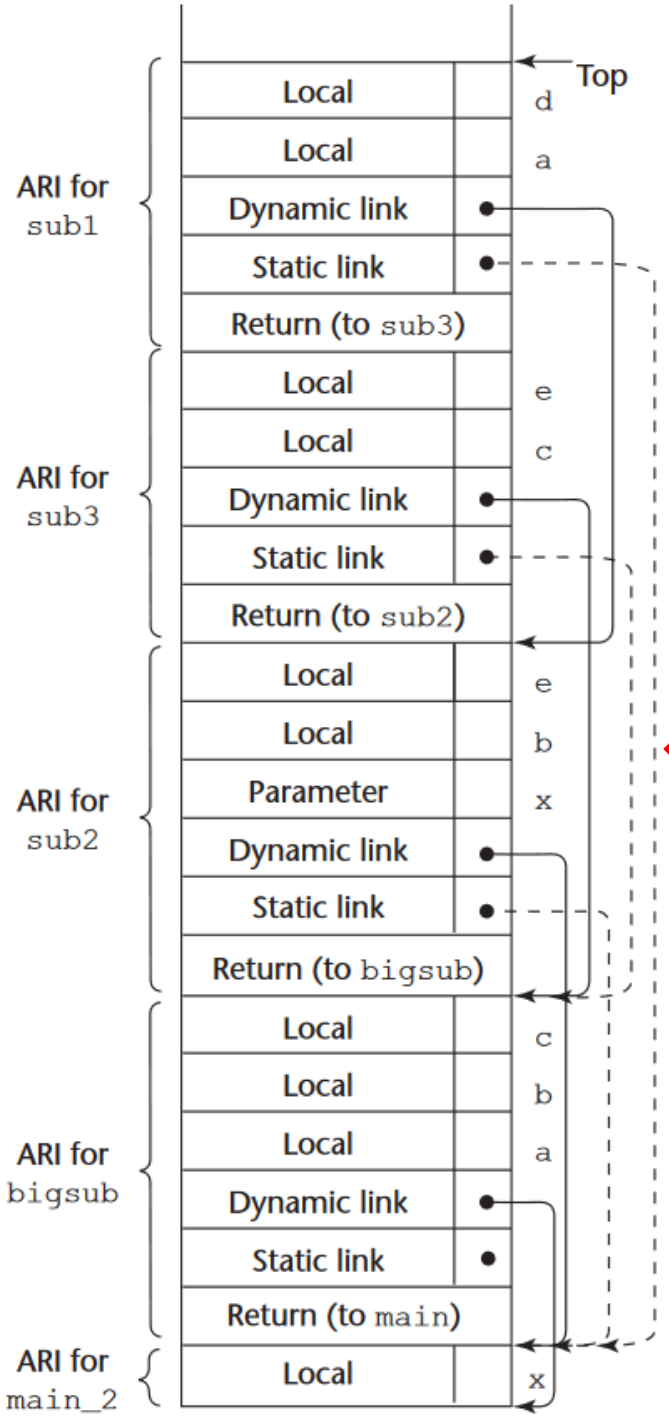


Esempio con funzioni annidate in JavaScript

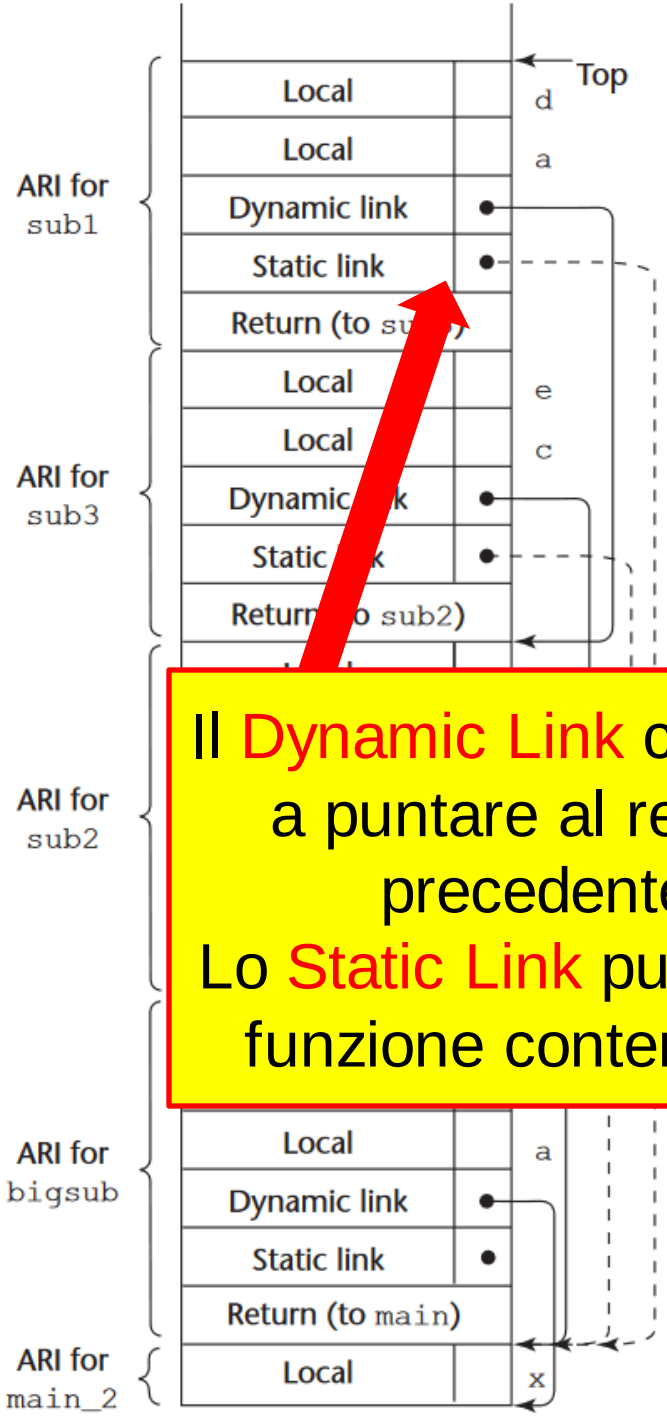
La sequenza delle chiamate:

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

```
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
```

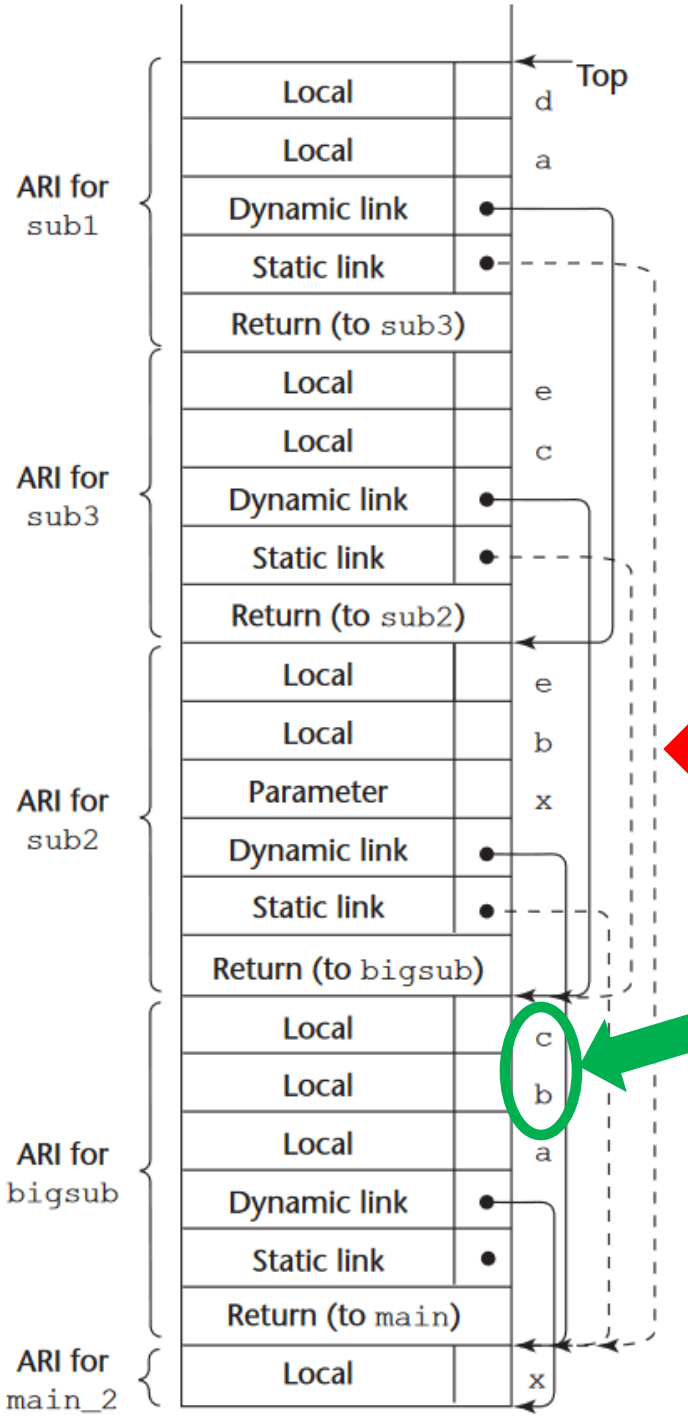



```
function main(){
  var x;
  function bigsub() {
    var a, b, c;
    function sub1 {
      var a, d;
      a = b + c; <---- eseguito 1^
    } // end of sub1
    function sub2(x) {
      var b, e;
      function sub3() {
        var c, e;
        sub1();
        e = b + a; <-- eseguito 2^
      }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



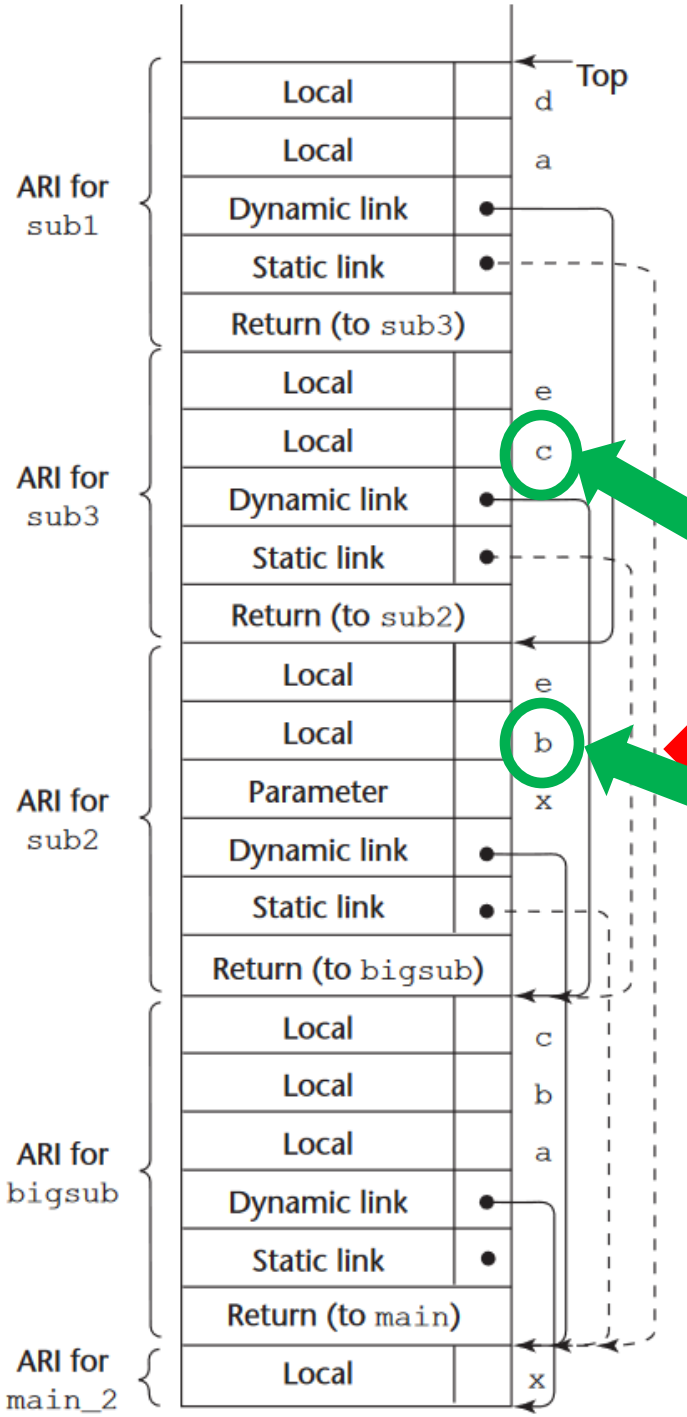
Il Dynamic Link continua a puntare al record precedente
Lo Static Link punta alla funzione contenitrice

```
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
                                    2^
                                    3^
    } // end of sub2
    sub2(7);
  } // end of bigsub
  bigsub();
} // end of main
```

la funzione `sub1` trova `b` e `c` seguendo lo Static Link (scoping statico)




```
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;
    }
    su
  } //
  bigs
} // end of main
```

Nei linguaggi con scoping dinamico (non JavaScript) non c'è lo Static Link e le variabili non locali si cercano scorrendo la pila, ossia seguendo il Dynamic Link.
(Trova altri `b` e `c`)

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 bindings

Ambiente

- Un ambiente Σ è una collezione di binding
- Esempio Σ = **{x -> 25, y -> 6}**
- 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 **6**
  - l'identificatore **z** non è legato nell'ambiente
- Astrattamente un ambiente è una funzione di tipo

  Ide → Value + Unbound
- L'uso della costante **Unbound** permette di rendere la funzione totale

Ambiente

- Dato un ambiente **Σ: Ide → Value + Unbound**
- **Σ(x)** denota il valore **v** associato a **x** nell'ambiente oppure il valore speciale **Unbound**
- **Σ[x=v]** indica l'ambiente esteso così definito
  - **Σ[x=v](y) = v** se **y = x**
  - **Σ[x=v](y) = Σ(y)** se **y != x**
- Esempio: se Σ = {x -> 25, y -> 7} allora

  Σ[x=5] = {x -> 5, y -> 7}

Implementazione (semplice, come lista di coppie)

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

(* 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)

```
(* 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)
```

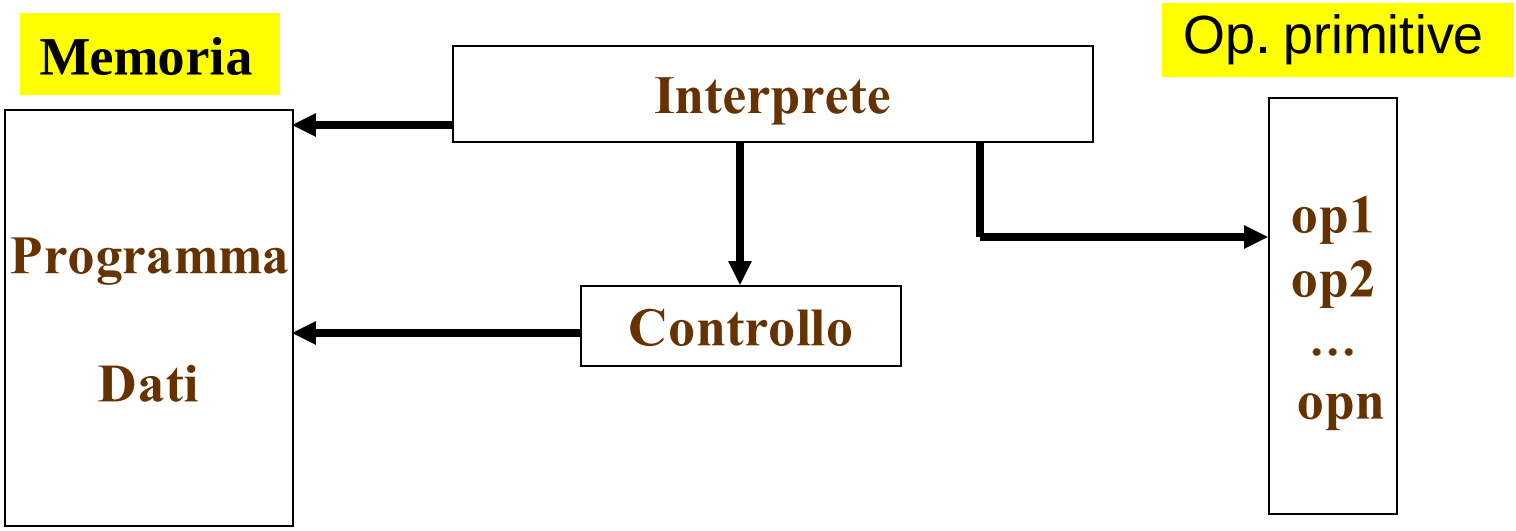
Linguaggio MiniCaml

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**
  - **del supporto a run time per il linguaggio**

Struttura dell'interprete

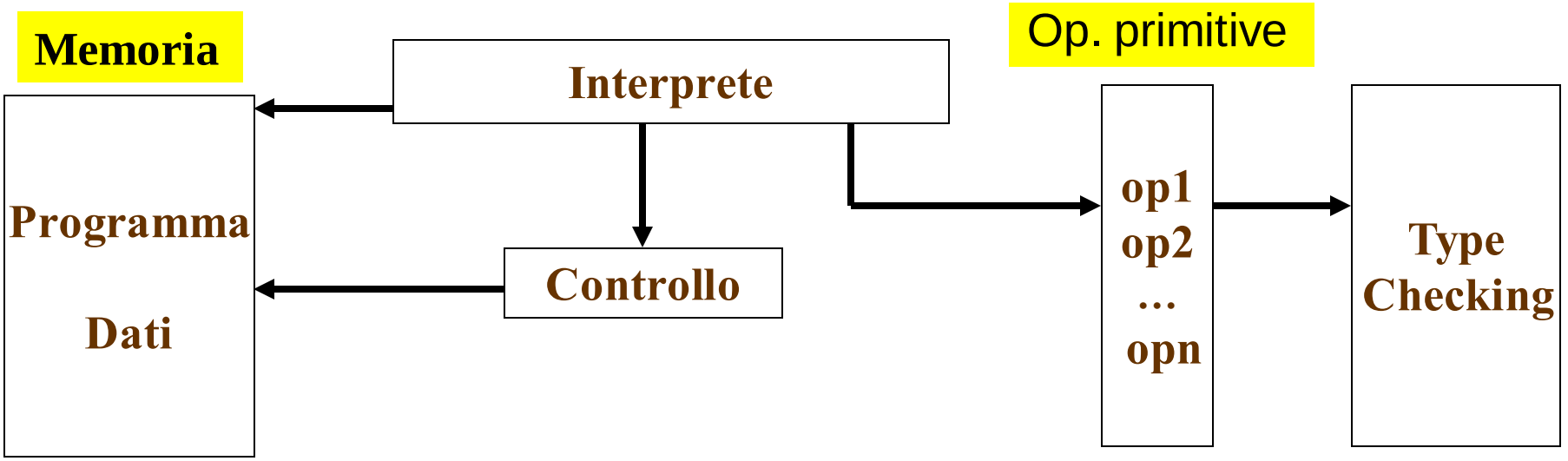
Seguiremo la struttura generale delle macchine astratte



Struttura dell'interprete

Seguiremo la struttura generale delle macchine astratte

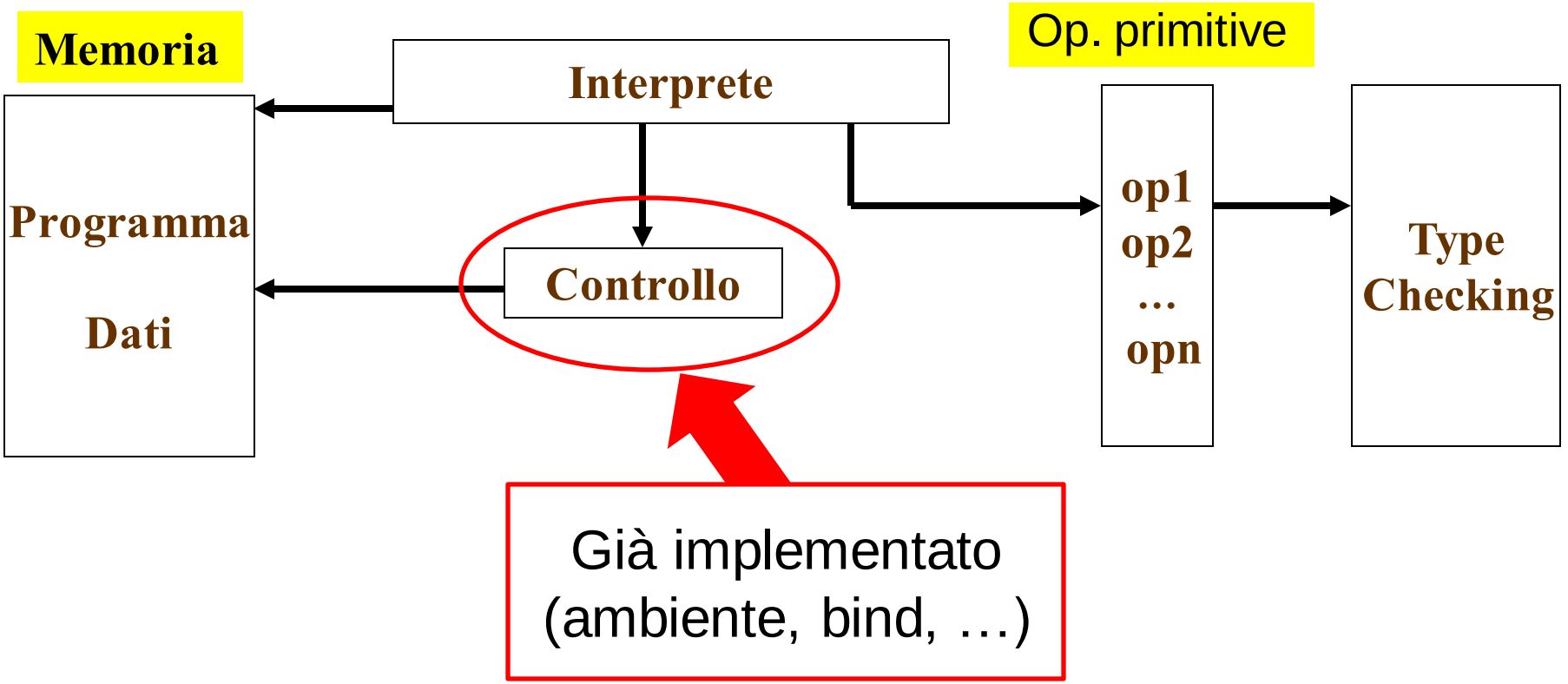
- Esplicitando le operazioni di type checking (dinamico)



Struttura dell'interprete

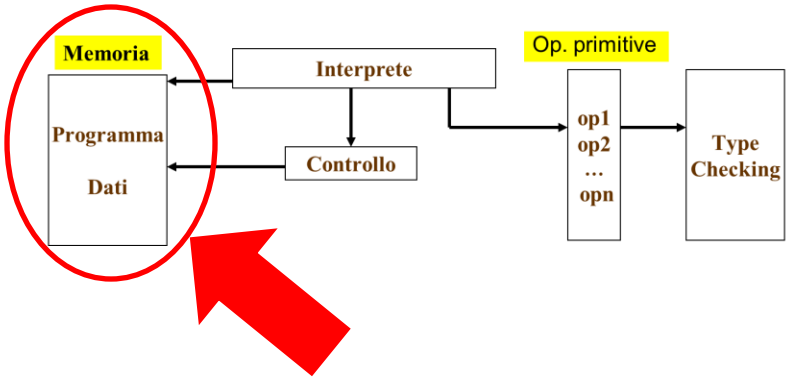
Seguiremo la struttura generale delle macchine astratte

- Esplicitando le operazioni di type checking (dinamico)



Linguaggio funzionale didattico

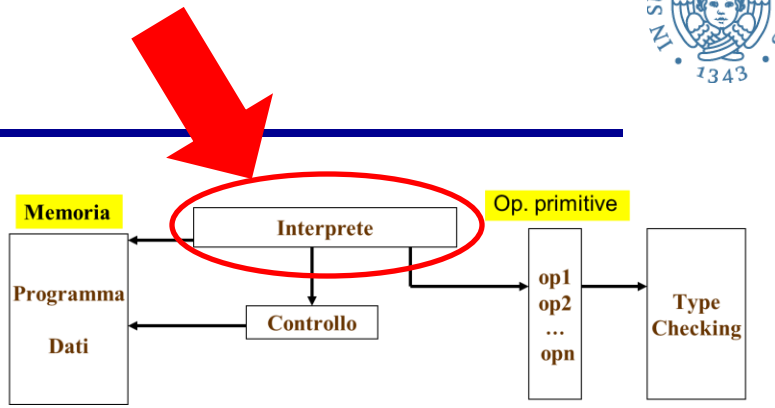
```
type ide = string
 type exp =
      | CstInt of int              (* Letterale n *)
      | CstTrue                    (* Letterale true *)
      | CstFalse                   (* Letterale false *)
      | Sum of exp * exp           (* Somma *)
      | Diff of exp * exp          (* Sottrazione *)
      | 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 *)
      | Den of ide                 (* Entità denotabile (variabile) *)
      | Ifthenelse of exp * exp * exp     (* Espressione condizionale *)
      | 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 *)
```



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 e ambiente

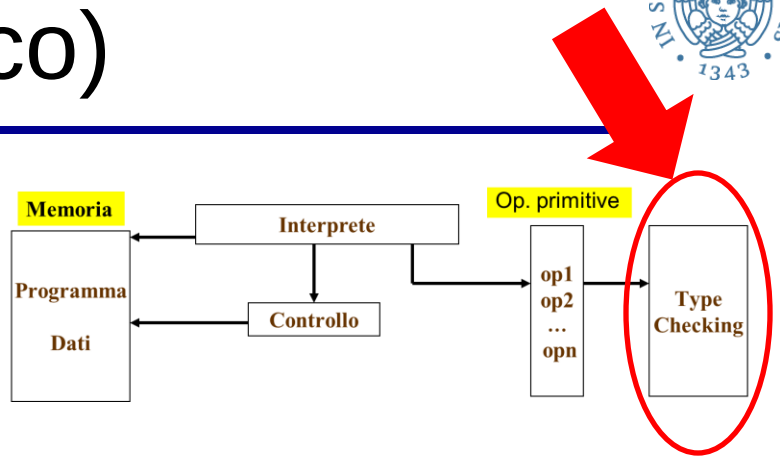
- **Valori esprimibili (risultato della valutazione di espressioni)**

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

DESCRITTORE DI TIPO

- **Ambiente: associazione ide evT env**
- **Nell'ambiente di un interprete i valori devono avere anche l'informazione sul tipo per consentire il type checking dinamico**

Typechecking (dinamico)

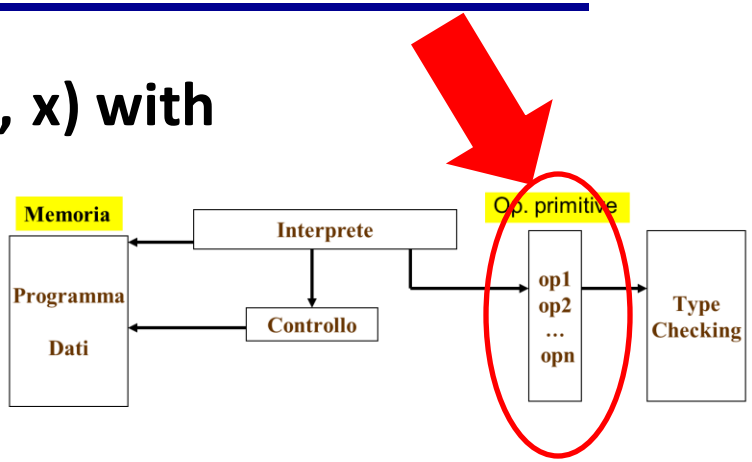



```
let typecheck (type, typeDescriptor) =
 match type with
  | TInt ->
      (match typeDescriptor with
       | Int(u) -> true
       | _ -> false)
  | TBool ->
      (match typeDescriptor with
       | Bool(u) -> true
       | _ -> false)
  | _ -> failwith ("not a valid type");;

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

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

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 ");;
```

Operazioni di base

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

Implementazione ops di base

```
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 ");;
```

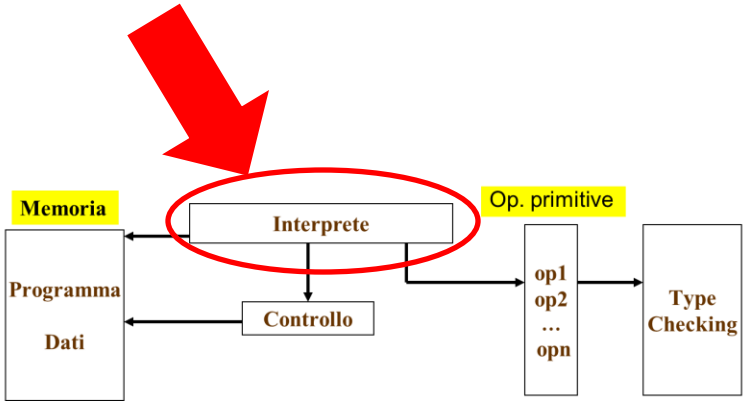
Operazioni di base

```
let is_zero x = match (typecheck(TInt,x), x) with
   | (true, In
   | (_, _) ->

let int_eq(x
  match (typ
     | (true, t
     | (_,_,_,_

let int_plus
match(type
   | (true, true, Int(v), Int(w)) -> Int(v + w)
   | (_,_,_,_) -> failwith("run-time error ");;
```

Le operazione di base sono implementate tramite una regola di valutazione eager:
prima di applicare l'operatore, si valutano tutti i sottoalberi (sottoespressioni)

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))
  | 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))
```



Binding identificatori

Regola semantica operazionale

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

Regola Interprete (nella funzione eval)

```
let rec eval (e:exp) (s: evT env) : evT =
 match e with
 | …
 | Den(i) -> s i
```

Condizionale: regole operazionali

SINTASSI ASTRATTA

Ifthenelse of exp * exp * exp

Regole operazionali

$$\frac{\Sigma \rhd cond \implies true \quad \Sigma \rhd e1 \implies v1}{\Sigma \rhd Ifthenelse(cond, e1, e2) \Rightarrow v1}$$

$$\frac{\Sigma \rhd cond \implies false \quad \Sigma \rhd e2 \implies v2}{\Sigma \rhd Ifthenelse(cond, e1, e2) \Rightarrow v2}$$

Condizionale: regola interprete

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

La valutazione del condizionale non segue una stragegia **eager**:
è l'operatore che richiede la valutazione dei **sottoalberi**, in base alla valutazione della **guardia**

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 (in parte) 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 operazionale del blocco

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

Semantica operazionale del blocco

Σ = run-time stack

push RA su Σ

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

Uscita blocco
pop su Σ

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

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 myp' = CstInt(3);;
val myp' : exp = CstInt 3

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

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

diventa

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

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

  Fun of ide * exp

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

  Apply of Den("f") * exp

- Per ora non ci occupiamo del modo del passaggio parametri
  - le espressioni parametro attuale sono valutate e i valori ottenuti legati nell'ambiente al corrispondente parametro formale
- Ignoriamo le funzioni ricorsive
- Assumiamo di avere funzione unarie
- Introducendo le funzioni, il linguaggio funzionale è completo
  - un linguaggio funzionale reale (tipo ML) ha in più i tipi, il pattern matching e le eccezioni

Analisi semantica

- Come bisogna estendere i tipi esprimibli (**evT**) per comprendere le astrazioni funzionali?
  - Quale è il valore di una funzione?
- Assumiamo **scoping statico** (vedremo poi quello **dinamico**)

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
  - nome del parametro formale (**ide**)
  - codice della funzione dichiarata (**exp**)
  - ambiente al momento della dichiarazione (**evT env**)

Scoping statico: **i riferimenti non locali dell'astrazione sono risolti nell'ambiente di dichiarazione della funzione**

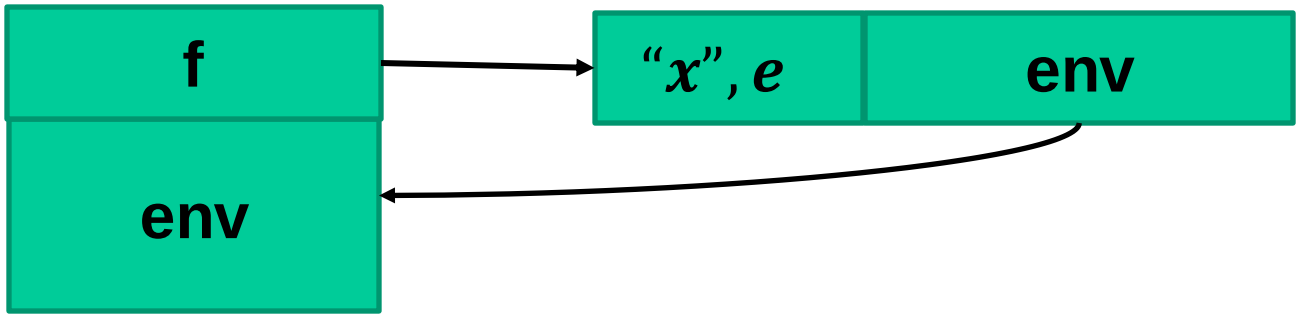
Astrazione e applicazione di funzione: regole operazionali (**scoping static)**

$$\Sigma \triangleright Fun(x, e) \Longrightarrow Closure(\text{“}x\text{”}, e, \Sigma)$$

$$\frac{\Sigma \triangleright Var(\text{”}f\text{”}) \Longrightarrow Closure(\text{“}x\text{”}, body, \Sigma_{fDecl}) \qquad \Sigma \triangleright \arg \Longrightarrow va \qquad \Sigma_{fDecl}[x = va] \triangleright body \Longrightarrow v}{\Sigma \triangleright Apply(Den(\text{“}f\text{”}), arg) \Longrightarrow v}$$

Dichiarazione di una funzione

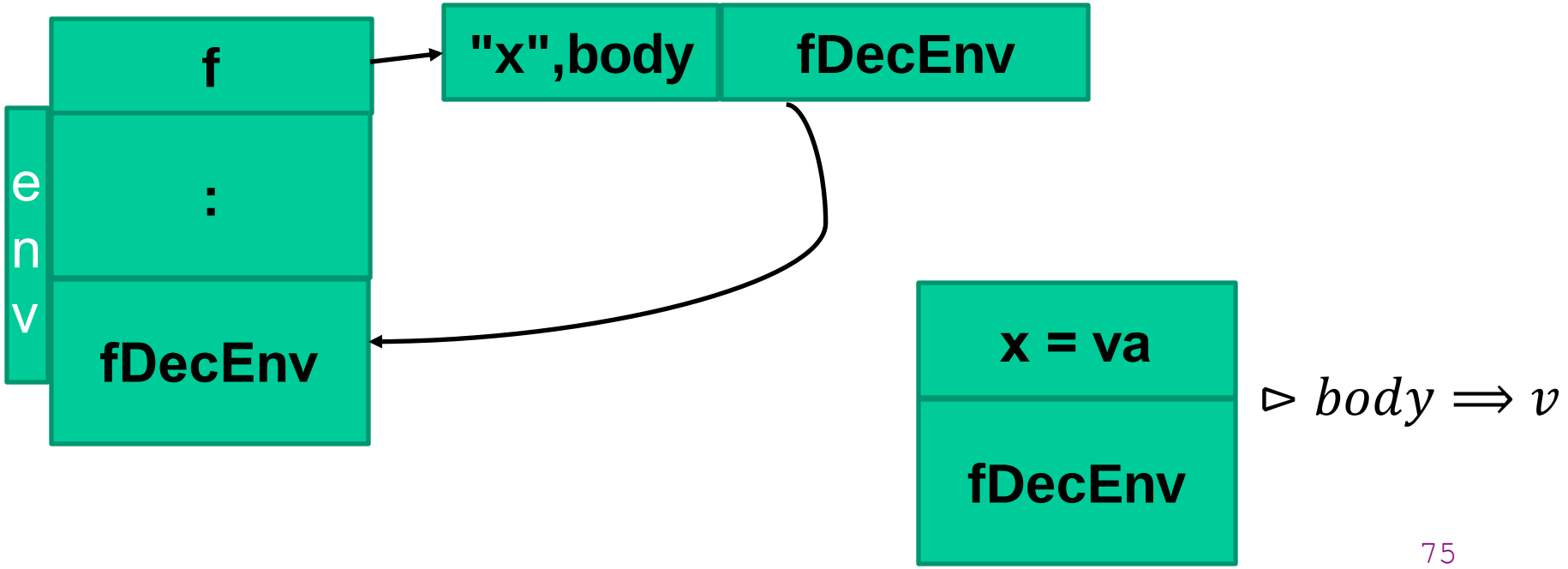
$$\frac{\Sigma \rhd Fun(x, e) \Longrightarrow Closure(\text{“}x\text{”}, e, \Sigma) \quad \Sigma[f = Closure(\text{“}x\text{”}, e, \Sigma)] \rhd e' \Longrightarrow v'}{\Sigma \rhd Let(\text{“}f\text{”}, Fun(x, e), e') \Longrightarrow v'}$$



Applicazione di una funzione

$$\Sigma \triangleright Den(\text{"}f\text{"}) \implies Closure(\text{“}x\text{”}, body, \Sigma_{fDecl})$$

$$\frac{\Sigma \triangleright \arg \implies va \qquad \Sigma_{fDecl}[x = va] \triangleright body \implies v}{\Sigma \triangleright Apply(Den(\text{“}f\text{”}), arg) \implies v}$$



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") ;;
```

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") ;;
```

Il corpo della funzione viene valutato nell'ambiente ottenuto legando il parametro formale al valore del parametro attuale nell'ambiente nel quale era stata valutata l'astrazione

Semantica operazionale vs. eseguibile

$$\frac{\Gamma \triangleright Den(\text{"}f\text{"}) \Longrightarrow Closure(\text{"}x\text{"}, body, \Gamma_{fDecl}) \qquad \Gamma \triangleright \mathbf{arg} \Longrightarrow va \qquad \Gamma_{fDecl}[x = va] \triangleright body \Longrightarrow v}{\Gamma \triangleright Apply(Den(\text{"}f\text{"}), arg) \Longrightarrow v}$$

```
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 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
```

Scoping Dinamico

- **Scoping dinamico**. Dobbiamo modificare **evT**

```
type evT =  | Int of int | Bool of bool | Unbound
            | Funval of efun
and efun = ide * exp
```

- La definizione di **efun** mostra che l'astrazione funzionale contiene solo il codice della funzione dichiarata
- Il corpo della funzione verrà valutato nell'ambiente ottenuto
  - legando i parametri formali ai valori dei parametri attuali
  - nell'ambiente nel quale avviene la applicazione

Astrazione e applicazione di funzione: **scoping dinamico**

Astrazione funzionale

$$\Sigma \triangleright Fun(\text{“}x\text{”}, e) \Longrightarrow Funval(\text{“}x\text{”}, e)$$

Applicazione

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

Scoping dinamico: interprete

```
| Fun(arg, ebody) -> Funval(arg,ebody)
| Apply(Den(f), eArg) ->
    let fval = s f in
      (match favl 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") ;;
```

Il corpo della funzione viene valutato nell'ambiente ottenuto legando i parametri formali ai valori dei parametri attuali nell'ambiente dove viene effettuata la chiamata

Ricapitolando: regole di scoping

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

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

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

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

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

Definizioni ricorsive

Funzioni ricorsive

- Come è fatta una definizione di funzione ricorsiva?
- E una espressione `Let(f, e1, e2)` nella quale
  - `f` è il nome della funzione (ricorsiva)
  - `e1` è un'astrazione `Fun(i, a)` nel cui corpo occorre una applicazione di `Den f`

Esempio

```
Let("fact",
    Fun("x", Ifthenelse(Eq(Den "x", Eint 0), Eint 1,
                        Prod(Den "x",
                             Appl(Den "fact",
                                  [Diff(Den "x", Eint 1)])))),
    Appl(Den "fact", [Eint 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 una associazione per il parametro formale **x**. Ma **s** non contiene legami per il nome **"fact"** pertanto **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 di
  - un diverso costrutto per "dichiarare" funzioni ricorsive (come il **let rec** di ML)
  - oppure 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

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"),Sun(Den("n"),CstInt(1))))),
Apply(Den("fact"),CstInt(3)))
```

Valori esprimibili evT

- Estendere i valori esprimibli (**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)
```

Il codice dell'interprete

```
:
| Letrec(f, i, fBody, letBody) ->
   let benv =
     bind s  f  (RecClosure(f, i, fBody, s))
      in eval letBody  benv
:
```

Viene associato al nome della funzione ricorsiva una chiusura ricorsiva che contiene il nome della funzione stessa

Il codice dell'interprete (2)

```
    :
| 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

- Il valore della chiusura ricorsiva **RecClosure** è recuperato dall'ambiente corrente
- Il parametro attuale è valutato nell'ambiente del chiamante ottenendo il valore **aVal**
- L'ambiente statico **fDecEnv**, memorizzato nella chiusura, è esteso con il legame tra il nome della funzione e la sua chiusura ricorsiva, ottendo l'ambiente **rEnv**
- L'ambiente effettivo di esecuzione **aEnv** si ottiene estendendo l'ambiente **rEnv** con il binding del passaggio del parametro

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 Function

- Estendiamo la sintassi del linguaggio didattico per avere al possibilità di trattare funzioni come valori di prima classe.
- Questo significa ammettere la possibilità che il risultato della valutazione di una espressione sia una funzione.

Higher Order Function: sintassi astratta

Estendiamo la sintassi astratta del linguaggio

exp ::= …. | Apply of exp * exp

Idea: l'applicazione funzionale **Apply(eF, eArg)** si ottiene

(1) valutando l'espressione **eF** ottenendo un valore funzionale,

(2) valutando il corpo della funzione (estratto dalla chiusura) nell'ambiente statico esteso con il legame tra il parametro formale e il valore del parametro attuale (**eArg**)

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")) ;;
```