



BLOCCHI E AMBIENTE LOCALE IN LINGUAGGI FUNZIONALI E IMPERATIVI

Contenuti



- ✎ nomi e associazioni: l'ambiente
- ✎ operazioni sulle associazioni
 - creazione e distruzione
 - disattivazione e riattivazione
- ✎ la struttura a blocchi nei linguaggi funzionali
 - semantica del let
 - operativa
 - interprete iterativo
 - ✓ attivazioni e records di attivazione
- ✎ la struttura a blocchi nei linguaggi imperativi
 - semantica delle dichiarazioni e memoria locale
 - operativa
 - interprete iterativo

Nomi e ambiente

- ✚ Identificatori = nomi simbolici per denotare vari tipi di entità
 - Costanti, nomi delle operazioni primitive, variabili, sottoprogrammi, parametri formali
- ✚ l'associazione tra nomi e oggetti denotati, è l'ambiente
 - principale differenza tra i linguaggi macchina ed i linguaggi ad alto livello
- ✚ può esistere un ambiente globale
 - associazioni comuni a diverse unità di programmi
 - ✓ create dal programma principale
 - ✓ esportate da un modulo
- ✚ esiste sempre un ambiente locale
 - associazioni create con due meccanismi
 - ✓ dichiarazioni all'ingresso in un blocco
 - ✓ passaggio di parametri in occasione di una chiamata di sottoprogramma



Ambiente locale dinamico

- ✚ *creazione di una associazione tra nome e oggetto denotato*
 - all'ingresso di un blocco
 - inserimento nell'ambiente
- ✚ *distruzione di una associazione*
 - all'uscita dal blocco in cui l'associazione è stata creata
 - eliminazione dall'ambiente
- ✚ l'associazione è utilizzabile solo all'interno del blocco
- ✚ se rientriamo nello stesso blocco
 - rieseguiamo le dichiarazioni
 - creiamo nuove associazioni



ambiente locale statico



- ✦ attivazione *di una associazione tra nome e oggetto denotato*
 - all'ingresso di un blocco, sua riattivazione nell'ambiente
- ✦ disattivazione *di una associazione*
 - all'uscita del blocco in cui l'associazione è stata attivata
 - sua disattivazione nell'ambiente
- ✦ l'associazione è utilizzabile solo all'interno del blocco
- ✦ se rientriamo nello stesso blocco
 - non rieseguiamo le dichiarazioni
 - ci limitiamo a riattivare le associazioni precedenti
- ✦ chi esegue le dichiarazioni?
- ✦ il compilatore, il collegatore, il caricatore (prima dell'inizio dell'esecuzione)
 - la prima esecuzione del blocco (o simile)

E la memoria?



- ✦ l'esecuzione di una dichiarazione
 - una volta, oppure
 - tante volte quante sono le esecuzioni del blocco
- ✦ può provocare
 - oltre alla creazione di una associazione nell'ambiente
 - anche la allocazione di *memoria locale*
- ✦ la memoria locale segue l'evoluzione dell'ambiente locale
 - se l'ambiente locale è dinamico, la memoria locale è dinamica
 - se l'ambiente locale è statico, la memoria locale è statica
- ✦ se l'ambiente locale è statico, la memoria locale ad un blocco viene preservata tra due diverse esecuzioni del blocco
 - costituisce una sorta di stato interno al blocco

Ambiente locale statico o dinamico?



- tutti i linguaggi moderni utilizzano l'ambiente locale dinamico
 - eventualmente alcune associazioni possono essere trattate in modo statico
- nel linguaggio didattico, assumiamo di avere l'ambiente locale dinamico
- vedremo poi cosa succede se l'ambiente locale è statico
 - incluse le sue versioni moderne

Il costrutto `let` nel linguaggio funzionale



```
type ide = string
type exp =
  | Eint of int
  | Ebool of bool
  | Den of ide
  | Prod of exp * exp
  | Sum of exp * exp
  | Diff of exp * exp
  | Eq of exp * exp
  | Minus of exp
  | Iszero of exp
  | Or of exp * exp
  | And of exp * exp
  | Not of exp
  | Ifthenelse of exp * exp * exp
  | Let of ide * exp * exp
```

Sintassi concreta e astratta



- 👁 In ML:
let x = 5 in X + 10;;
- 👁 Nel linguaggio didattico (sintassi astratta)
let(x, Eint 5, sum(den x, Eint 10))

Perché i blocchi?



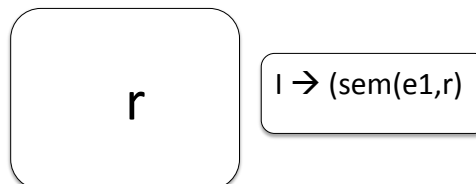
- 👁 con il let possiamo cambiare l'ambiente in punti arbitrari all'interno di una espressione
 - facendo sì che l'ambiente "nuovo" valga soltanto durante la valutazione del "corpo del blocco"
 - lo stesso nome può denotare entità distinte in blocchi diversi
- 👁 i blocchi possono essere annidati
 - e l'ambiente locale di un blocco più esterno può essere (in parte) visibile ed utilizzabile nel blocco più interno
 - ✓ come ambiente non locale!
- 👁 La nozione di blocco porta naturalmente a
 - ✓ una semplice gestione dinamica della memoria locale
 - si sposa felicemente con la regola di scoping statico
 - ✓ per la gestione dell'ambiente non locale

La semantica operativa

```
let rec sem ((e:exp), (r:eval env)) =  
  match e with  
  | Eint(n) -> Int(n)  
  | Ebool(b) -> Bool(b)  
  | Den(i) -> applyenv(r,i)  
  | Iszero(a) -> iszero(sem(a, r))  
  | Eq(a,b) -> equ(sem(a, r),sem(b, r))  
  | Prod(a,b) -> mult(sem(a, r), sem(b, r))  
  | Sum(a,b) -> plus(sem(a, r), sem(b, r))  
  | Diff(a,b) -> diff(sem(a, r), sem(b, r))  
  | Minus(a) -> minus(sem(a, r))  
  | And(a,b) -> et(sem(a, r), sem(b, r))  
  | Or(a,b) -> vel(sem(a, r), sem(b, r))  
  | Not(a) -> non(sem(a, r))  
  | Ifthenelse(a,b,c) -> let g = sem(a, r) in  
    if typecheck("bool",g) then  
      (if g = Bool(true) then sem(b, r) else sem(c, r))  
    else failwith ("nonboolean guard")  
  | Let(i,e1,e2) -> sem(e2, bind (r ,i, sem(e1, r)))  
  
val sem : exp * eval Funenv.env -> eval = <fun>
```



**Let(I,e1,e2) ->
 sem(e2, bind (r ,I, sem(e1, r)))**



Semantica del let: commenti

```
let rec sem ((e:exp), (r:eval env)) =  
  match e with  
  ....  
  | Let(i,e1,e2) -> sem(e2, bind (r ,i, sem(e1, r)))
```



- ✎ l'espressione e2
 - corpo del blocco
 - è valutata nell'ambiente "esterno" r esteso con l'associazione tra il nome i ed il valore di e1
- ✎ associazioni per nomi diversi da i eventualmente presenti in r sono disponibili (visibili) durante la valutazione di e2

Esempio

- ✎ Let x = 1 in
 let y = if x = 0 then x-1 else x+1 in
 let z = x+y in z
- ✎ R = empty

{	}	X	Y	Z
		1	2	3



Esempio eseguibile



```
sem(Let("x", Sum(Eint 1, Eint 0),
  Let("y", Ifthenelse(Eq(Den "x", Eint 0),
    Diff(Den "x", Eint 1), Sum(Den "x", Eint 1)),
  Let("z", Sum(Den "x", Den "y"), Den "z"))
  , emptyenv Unbound;;
eval = Int 3
```

Come eliminiamo la ricorsione



- ✎ la funzione ricorsiva sem ha due argomenti (l'espressione, l'ambiente) e calcola un eval
- ✎ in presenza del let
 - l'ambiente viene modificato nelle chiamate ricorsive di sem
 - e "ripristinato" al ritorno dalla chiamata ricorsiva (uscita dal blocco)
- ✎ si può gestire con una pila di ambienti, su cui
 - si effettua l'operazione di push all'entrata nel blocco
 - si effettua il pop all'uscita dal blocco
- ✎ Abbiamo bisogno di una pila di ambienti

Domanda?



- ✎ Come interagisce la pila degli ambienti, con le altre pile introdotte per gestire le espressioni ed i loro risultati intermedi?
 - per ogni ambiente, serve una coppia di pile `continuation` e `tempstack`
- ✎ tre pile
 - `envstack` pila di ambienti
 - `cstack` pila di pile di espressioni etichettate
 - `tempvalstack` pila di pile di eval

L'attivazione ed il "record di attivazione"



- ✎ con l'ambiente locale dinamico, ogni volta che si entra in un blocco si crea una nuova *attivazione*
 - i blocchi sono un caso particolare di sottoprogrammi
 - ✓ senza parametri
- ✎ ad una attivazione corrisponde nell'implementazione iterativa la creazione di un nuovo *record di attivazione*
 - che contiene tutte le informazioni caratteristiche della attivazione
 - ✓ ambiente
 - ✓ espressione da valutare e struttura dati (pila) necessaria per farlo
 - ✓ struttura dati (pila) per memorizzare i valori temporanei

Recod attivazione



- nelle implementazioni standard, esiste un'unica *pila dei record di attivazione*
 - un record viene inserito sulla pila all'entrata nel blocco
 - un record viene eliminato dalla pila all'uscita dal blocco
- Nella nostra implementazione didattica invece di un'unica pila dei records di attivazione, tre pile gestite in modo "parallelo"
 - `envstack` pila di ambienti
 - `cstack` pila di pile di espressioni etichettate
 - `tempvalstack` pila di pile di eval

Env-stack C-stack Temp-stack



Le strutture dell'interprete iterativo

```
let cframesize(e) = 20
let tframesize(e) = 20
let stacksize = 100

type labeledconstruct =
  | Expr1 of exp
  | Expr2 of exp

let (cstack: labeledconstruct stack stack) = emptystack(stacksize, emptystack(1, Expr1(Eint(0))))

let (tempvalstack: eval stack stack) = emptystack(stacksize, emptystack(1, Unbound))

let (envstack: eval env stack) = emptystack(stacksize, emptyenv(Unbound))

let pushenv(r) = push(r, envstack)

let topenv() = top(envstack)

let svuotaenv() = svuota(envstack)

let popenv () = pop(envstack)
```



L'interprete iterativo 1

la creazione di un nuovo record di attivazione (frame)

```
let newframes(e, rho) =
  let cframe = emptystack(cframesize(e), Expr1(e)) in
  let tframe = emptystack(tframesize(e), Unbound) in
  push(Expr1(e), cframe);
  push(cframe, cstack);
  push(tframe, tempvalstack);
  pushenv(rho)
```



L'interprete iterativo 2

```

let sem ((e:exp), (r:eval env)) =
  push(emptystack(1,Unbound),tempvalstack);
  newframes(e,r);
  while not(empty(cstack)) do
    while not(empty(top(cstack))) do
      let continuation = top(cstack) in
      let tempstack = top(tempvalstack) in
      let rho = topenv() in
      (match top(continuation) with
      | Expr1(x) ->
        (pop(continuation); push(Expr2(x),continuation);
        (match x with
        | Iszero(a) -> push(Expr1(a),continuation)
        | Eq(a,b) -> push(Expr1(a),continuation); push(Expr1(b),continuation)
        | Prod(a,b) -> push(Expr1(a),continuation); push(Expr1(b),continuation)
        | Sum(a,b) -> push(Expr1(a),continuation); push(Expr1(b),continuation)
        | Diff(a,b) -> push(Expr1(a),continuation); push(Expr1(b),continuation)
        | Minus(a) -> push(Expr1(a),continuation)
        | And(a,b) -> push(Expr1(a),continuation); push(Expr1(b),continuation)
        | Or(a,b) -> push(Expr1(a),continuation); push(Expr1(b),continuation)
        | Not(a) -> push(Expr1(a),continuation)
        | Ifthenelse(a,b,c) -> push(Expr1(a),continuation)
        | Let(i,e1,e2) -> push(Expr1(e1),continuation)
        | _ -> ()))

```



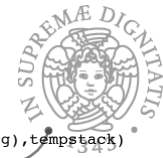
L'interprete iterativo 3

```

| Expr2(x) -> (pop(continuation); (match x with
| Eint(n) -> push(Int(n),tempstack)
| Ebool(b) -> push(Bool(b),tempstack)
| Den(i) -> push(applyenv(rho,i),tempstack)
| Iszero(a) -> let arg=top(tempstack) in pop(tempstack); push(iszero(arg),tempstack)
| Eq(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
  let sndarg=top(tempstack) in pop(tempstack); push(equ(firstarg,sndarg),tempstack)
| Prod(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
  let sndarg=top(tempstack) in pop(tempstack); push(mult(firstarg,sndarg),tempstack)
| Sum(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
  let sndarg=top(tempstack) in pop(tempstack); push(plus(firstarg,sndarg),tempstack)
| Diff(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
  let sndarg=top(tempstack) in pop(tempstack); push(diff(firstarg,sndarg),tempstack)
| Minus(a) -> let arg=top(tempstack) in pop(tempstack); push(minus(arg),tempstack)
| And(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
  let sndarg=top(tempstack) in pop(tempstack); push(et(firstarg,sndarg),tempstack)
| Or(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
  let sndarg=top(tempstack) in pop(tempstack); push(vel(firstarg,sndarg),tempstack)
| Not(a) -> let arg=top(tempstack) in pop(tempstack); push(non(arg),tempstack)

| Let(i,e1,e2) -> let arg=top(tempstack) in pop(tempstack);
  newframes(e2, bind(rho, i, arg))
| Ifthenelse(a,b,c) -> let arg=top(tempstack) in pop(tempstack);
  if typecheck("bool",arg) then
    (if arg = Bool(true) then push(Expr1(b),continuation)
    else push(Expr1(c),continuation)) else failwith ("type error"))))
done;
let valore= top(top(tempvalstack)) in pop(top(tempvalstack));
popenv(); pop(cstack); pop(tempvalstack); push(valore,top(tempvalstack));
done;
let valore= top(top(tempvalstack)) in pop(top(tempvalstack)); pop(tempvalstack); valore;;
val sem : exp * eval Funenv.env -> eval = <fun>

```



Blocchi in un linguaggio imperativo



- ✦ un blocco consiste in
 - una lista di dichiarazioni seguita da
 - una lista di comandi
- ✦ la lista di comandi “viene eseguita” nello stato (ambiente e memoria) risultante dall’esecuzione della lista di dichiarazioni
- ✦ il blocco è un comando
 - restituisce la memoria “esterna” (eventualmente) modificata dall’esecuzione della lista di comandi
 - l’ambiente “esterno” non viene modificato
 - le associazioni locali (e le locazioni di memoria eventualmente ad esse associate) esistono solo all’interno del blocco
- ✦ le dichiarazioni di variabili sono realizzate con un nuovo costrutto di tipo espressione (simile al `ref` di ML)

Linguaggio imperativo con blocchi: domini sintattici



```
type ide = string
type exp = | Eint of int
          | Ebool of bool
          | Den of ide
          | Prod of exp * exp
          | Sum of exp * exp
          | Diff of exp * exp
          | Eq of exp * exp
          | Minus of exp
          | Iszero of exp
          | Or of exp * exp
          | And of exp * exp
          | Not of exp
          | Ifthenelse of exp * exp * exp
          | Val of exp
          | Let of ide * exp * exp
          | Newloc of exp
type decl = (ide * exp) list
type com = | Assign of exp * exp
          | Cifthenelse of exp * com list * com list
          | While of exp * com list
          | Block of decl * com list
```

Semantica operativa: espressioni

```
let rec sem ((e:exp), (r:dval env), (s: mval store)) =
  match e with
  | Eint(n) -> Int(n)
  | Ebool(b) -> Bool(b)
  | Den(i) -> dvaltoeval(applyenv(r,i))
  | Iszero(a) -> iszero(sem(a, r, s))
  | Eq(a,b) -> equ(sem(a, r, s), sem(b, r, s))
  | Prod(a,b) -> mult (sem(a, r, s), sem(b, r, s))
  | Sum(a,b) -> plus (sem(a, r, s), sem(b, r, s))
  | Diff(a,b) -> diff (sem(a, r, s), sem(b, r, s))
  | Minus(a) -> minus(sem(a, r, s))
  | And(a,b) -> et (sem(a, r, s), sem(b, r, s))
  | Or(a,b) -> vel (sem(a, r, s), sem(b, r, s))
  | Not(a) -> non(sem(a, r, s))
  | Ifthenelse(a,b,c) ->
    let g = sem(a, r, s) in
    if typecheck("bool",g) then
      (if g = Bool(true)
       then sem(b, r, s)
       else sem(c, r, s))
    else failwith ("nonboolean guard")

  | Let(i,e1,e2) -> let (v, s1) = semden(e1, r, s) in sem(e2, bind (r, i, v), s1)
  | Val(e) -> let (v, s1) = semden(e, r, s) in (match v with
    | Dloc n -> mvaltoeval(applystore(s1, n))
    | _ -> failwith("not a variable"))

and semden ((e:exp), (r:dval env), (s: mval store)) = match e with
  | Den(i) -> (applyenv(r,i), s)
  | Newloc(e) -> let m = evaltomval(sem(e, r, s)) in let (l, s1) = allocate(s, m) in (Dloc l, s1)
  | _ -> (evaltodval(sem(e, r, s)), s)
val sem : exp * dval Funenv.env * mval Funstore.store -> eval = <fun>
val semden : exp * dval Funenv.env * mval Funstore.store -> (dval * mval Funstore.store) = <fun>
```

Semantica operativa: comandi

```
let rec semc((c: com), (r:dval env), (s: mval store)) = match c with
  | Assign(e1, e2) -> let (v1, s1) = semden(e1, r, s) in
    (match v1 with
     | Dloc(n) -> update(s1, n, evaltomval(sem(e2, r, s)))
     | _ -> failwith ("wrong location in assignment"))
  | Cifthenelse(e, c11, c12) -> let g = sem(e, r, s) in
    if typecheck("bool",g) then
      (if g = Bool(true) then semc1(c11, r, s) else semc1 (c12, r, s))
      else failwith ("nonboolean guard")
  | While(e, c1) -> let g = sem(e, r, s) in
    if typecheck("bool",g) then
      (if g = Bool(true) then semc1((c1 @ [While(e, c1)]), r, s)
       else s)
      else failwith ("nonboolean guard")
  | Block(b) -> semb(b, r, s)

and semc1(c1, r, s) = match c1 with
  | [] -> s
  | c::c11 -> semc1(c11, r, semc(c, r, s))

val semc : com * dval Funenv.env * mval Funstore.store -> mval Funstore.store = <fun>
val semc1 : com list * dval Funenv.env * mval Funstore.store -> mval Funstore.store = <fun>
```

Semantica operativa: dichiarazioni



```
and semb ((dl, cl), r, s) =
  let (r1, s1) = semdv(dl, r, s) in semcl(cl, r1, s1)

and semdv(dl, r, s) =
  match dl with
  | [] -> (r,s)
  | (i,e)::dll -> let (v, s1) = semden(e, r, s) in
    semdv(dll, bind(r, i, v), s1)

val semdv : decl * dval Funenv.env * mval Funstore.store ->
  dval Funenv.env * mval Funstore.store = <fun>
val semb : (decl * com list) * dval Funenv.env * mval Funstore.store -> mval
  Funstore.store = <fun>
```

Semantica del blocco: commenti



```
and semb ((dl, cl), r, s) =
  let (r1, s1) = semdv(dl, r, s) in semcl(cl, r1,
  s1)
```

- ✎ la lista di comandi `cl` è valutata
 - nell'ambiente "esterno" `r` esteso con la semantica delle dichiarazioni `dl`
 - nella memoria "esterna" `s` estesa con la semantica delle dichiarazioni `dl`
- ✎ la memoria restituita contiene anche ciò che è stato prodotto nel blocco
 - ma le locazioni nuove non sono accessibili dall'ambiente "esterno" `r`

Eliminare la ricorsione per le dichiarazioni

- ☛ il dominio delle dichiarazioni è già iterativo (tail recursive)

```
type decl = (ide * exp) list
```

- ☛ come per i comandi, si può utilizzare la struttura sintattica (lista di coppie) per mantenere l'informazione su quello che si deve ancora valutare

- basta una unica cella
 - ✓ per ogni attivazione
- che può essere "integrata" nella pila "locale" di costrutti sintattici etichettati

I records di attivazione dell'interprete iterativo

- ☛ come nel caso del linguaggio funzionale, ma ...
 - il record di attivazione deve contenere anche la memoria del blocco
- ☛ ad una attivazione corrisponde nell'implementazione iterativa la creazione di un nuovo *record di attivazione*
 - che contiene tutte le informazioni caratteristiche della attivazione
 - ✓ ambiente
 - ✓ costruito sintattico da valutare e struttura dati (pila) necessaria per farlo
 - ✓ strutture dati (pile) per memorizzare i valori temporanei (eval e dval)
 - ✓ memoria
- ☛ le pile "parallele" che realizzano la pila dei records di attivazione
 - envstack pila di ambienti
 - cstack pila di pile di costrutti sintattici etichettati
 - tempvalstack pila di pile di eval
 - tempdvalstack pila di pile di dval
 - storestack pila di memorie
- ☛ cosa succede quando si "esce da una attivazione"?

Uscire da un blocco



- ✎ il ciclo dell'interprete iterativo tratta in modo simile tutti i costrutti che provocano la creazione di una nuova attivazione (frame)
 - o espressioni (Let)
 - o comandi (Block)
- ✎ la distruzione dell'attivazione richiede che vengano "esportate" all'attivazione precedente cose diverse
 - o un `eval` se si trattava di un'espressione
 - o uno `store` se si trattava di un comando
- ✎ un problema simile si pone sull'attivazione iniziale, che può anche corrispondere ad una dichiarazione
 - o che deve restituire una coppia `env * store`
- ✎ una ulteriore informazione nel record di attivazione
 - o il costrutto sintattico che l'ha originata
- ✎ di conseguenza una ulteriore pila "parallela"
 - o `labelstack` pila di costrutti sintattici etichettati

Le strutture dell'interprete iterativo 1



```
let cframesize(e) = 20
let tframesize(e) = 20
let tframesize(e) = 20
let stacksize = 100
```

```
type labeledconstruct =
| Expr1 of exp
| Expr2 of exp
| Exprd1 of exp
| Exprd2 of exp
| Com1 of com
| Com2 of com
| Com1 of labeledconstruct list
| Dec1 of ide * exp
| Dec2 of ide * exp
| Dec1 of labeledconstruct list
```

```
let (cstack: labeledconstruct stack stack) = emptystack(stacksize, emptystack(1, Expr1(Eint(0))))
```

```
let (tempvalstack: eval stack stack) = emptystack(stacksize, emptystack(1, Novalue))
```

```
let (tempdvalstack: dval stack stack) = emptystack(stacksize, emptystack(1, Unbound))
```

```
let envstack = emptystack(stacksize, (emptyenv Unbound))
```

```
let storestack = emptystack(stacksize, (emptystore Undefined))
```

```
let (labelstack: labeledconstruct stack) = emptystack(stacksize, Expr1(Eint(0)))
```

Le strutture dell'interprete iterativo 2

```
let labelcom (dl: com list) = let dlr = ref(dl) in
  let ldlr = ref([]) in
  while not (!dlr = []) do
    let i = List.hd !dlr in
    ldlr := !ldlr @ [Coml(i)];
    dlr := List.tl !dlr
  done;
  Coml(!ldlr)

let labeldec (dl: (ide * exp) list) = let dlr = ref(dl) in
  let ldlr = ref([]) in
  while not (!dlr = []) do
    let i = List.hd !dlr in
    ldlr := !ldlr @ [Decl(i)];
    dlr := List.tl !dlr
  done;
  Decl(!ldlr)

let pushenv(r) = push(r,envstack)
let topenv() = top(envstack)
let popenv () = pop(envstack)
let svuotaenv() = svuota(envstack)
let pushstore(s) = push(s,storestack)
let popstore () = pop(storestack)
let svuotastore () = svuota(storestack)
let topstore() = top(storestack)
```



L'interprete iterativo 0

la creazione di un nuovo record di attivazione (frame)

```
let newframes(ss, rho, sigma) =
  pushenv(rho);
  pushstore(sigma);
  let cframe = emptystack(cframesize(ss),Expr1(Eint 0)) in
  let tframe = emptystack(tframesize(ss),Noval) in
  let dframe = emptystack(tdframesize(ss),Unbound) in
  push(ss, cframe);
  push(ss, labelstack);
  push(cframe,cstack);
  push(dframe,tempdvalstack);
  push(tframe,tempvalstack)
val newframes : labeledconstruct * dval Funenv.env *
  mval Funstore.store -> unit = <fun>
```



L'interprete iterativo 1



```
let itsem() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  (match top(continuation) with
  | Expr1(x) ->
    (pop(continuation); push(Expr2(x), continuation);
    (match x with
    | Iszero(a) -> push(Expr1(a), continuation)
    | Eq(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Prod(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Sum(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Diff(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Minus(a) -> push(Expr1(a), continuation)
    | And(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Or(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Not(a) -> push(Expr1(a), continuation)
    | Ifthenelse(a,b,c) -> push(Expr1(a), continuation)
    | Val(a) -> push(Expr1(a), continuation)
    | Newloc(e) -> failwith ("nonlegal expression for sem")
    | Let(i,e1,e2) -> push(Expr1(e1), continuation)
    | _ -> ()))
```

L'interprete iterativo 2



```
| Expr2(x) ->
  (pop(continuation); (match x with
  | Int(n) -> push(Int(n), tempstack)
  | Bool(b) -> push(Bool(b), tempstack)
  | Den(i) -> push(applenv(rho,i), tempstack)
  | Iszero(a) -> let arg=top(tempstack) in pop(tempstack); push(iszero(arg), tempstack)
  | Eq(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
    let sndarg=top(tempstack) in pop(tempstack); push(egu(firstarg, sndarg), tempstack)
  | Prod(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
    let sndarg=top(tempstack) in pop(tempstack); push(mult(firstarg, sndarg), tempstack)
  | Sum(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
    let sndarg=top(tempstack) in pop(tempstack); push(plus(firstarg, sndarg), tempstack)
  | Diff(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
    let sndarg=top(tempstack) in pop(tempstack); push(diff(firstarg, sndarg), tempstack)
  | Minus(a) -> let arg=top(tempstack) in pop(tempstack); push(minus(arg), tempstack)
  | And(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
    let sndarg=top(tempstack) in pop(tempstack); push(et(firstarg, sndarg), tempstack)
  | Or(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
    let sndarg=top(tempstack) in pop(tempstack); push(vel(firstarg, sndarg), tempstack)
  | Not(a) -> let arg=top(tempstack) in pop(tempstack); push(non(arg), tempstack)
  | Ifthenelse(a,b,c) -> let arg=top(tempstack) in pop(tempstack);
    if typecheck("bool", arg) then
      (if arg = Bool(true) then push(Expr1(b), continuation)
       else push(Expr1(c), continuation)
       else failwith ("type error"))))
  | Val(e) -> let v = top(tempdstack) in pop(tempdstack);
    (match v with
    | Dloc n -> push(mvaltoeval(applystore(!globalstore, n)), tempstack)
    | _ -> failwith("not a variable"))
  | Let(i,e1,e2) -> let arg= top(tempdstack) in
    pop(tempdstack); newframes(Expr1(e2), bind(rho, i, arg), sigma)
  | _ -> failwith("no more cases for itsem"))
  | _ -> failwith("no more cases for itsem"))

val itsem : unit -> unit = <fun>
```

L'interprete iterativo 3



```
let itsemnden() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  (match top(continuation) with
  | Exprd1(x) -> (pop(continuation); push(Exprd2(x), continuation);
    match x with
    | Den i -> ()
    | Newloc(e) -> push(Expr1(e), continuation)
    | _ -> push(Expr1(x), continuation))
  | Exprd2(x) -> (pop(continuation); match x with
    | Den i -> push(applyenv(rho,i), tempdstack)
    | Newloc(e) -> let m=evaltomval(top(tempstack)) in pop(tempstack);
      let (l, s1) = allocate(sigma, m) in push(Dloc l, tempdstack);
      popstore(); pushstore(s1)
    | _ -> let arg = top(tempstack) in pop(tempstack);
      push(evaltodval(arg), tempdstack)
    | _ -> failwith("No more cases for semden")
  )
  )
val itsemnden : unit -> unit = <fun>
```

L'interprete iterativo 4



```
let itsemcl () =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  let cl = (match top(continuation) with
  | Com1(d1) -> d1
  | _ -> failwith("impossible in semdecl")) in
  if cl = [] then pop(continuation) else
  (let currc = List.hd cl in let newcl = List.tl cl in pop(continuation); push(Com1(newcl), continuation);
  (match currc with
  | Com1(Assign(e1, e2)) -> pop(continuation); push(Com1(Com2(Assign(e1, e2))::newcl), continuation);
    push(Exprd1(e1), continuation); push(Expr1(e2), continuation)
  | Com2(Assign(e1, e2)) -> let arg2 = evaltomval(top(tempstack)) in pop(tempstack);
    let arg1 = top(tempdstack) in pop(tempdstack); (match arg1 with
    | Dloc(n) -> popstore(); pushstore(update(sigma, n, arg2))
    | _ -> failwith ("wrong location in assignment"))
  | Com1(While(e, cl)) -> pop(continuation); push(Com1(Com2(While(e, cl))::newcl), continuation);
    push(Expr1(e), continuation)
  | Com2(While(e, cl)) -> let g = top(tempstack) in pop(tempstack);
    if typecheck("bool",g) then (if g = Bool(true) then (let old = newcl in let newl =
      (match labelcom cl with
      | Com1 newl1 -> newl1
      | _ -> failwith("impossible in while")) in
      let nuovo = Com1(newl @ [Com1(While(e, cl)]) @ old) in pop(continuation); push(nuovo, continuation))
    else ()) else failwith ("nonboolean guard")
```

L'interprete iterativo 5



```
| Com1(Cifthenelse(e, c11, c12)) -> pop(continuation);
  push(Com1(Com2(Cifthenelse(e, c11, c12))::newcl), continuation); push(Expr1(e), continuation)
| Com2(Cifthenelse(e, c11, c12)) -> let g = top(tempstack) in pop(tempstack);
  if typecheck("bool",g) then (let temp = if g = Bool(true) then
    labelcom (c11) else labelcom (c12) in let new1 = (match temp with
      | Com1 new11 -> new11
      | _ -> failwith("impossible in cifthenelse")) in
    let nuovo = Com1(new1 @ newcl) in pop(continuation); push(nuovo, continuation))
  else failwith ("nonboolean guard")
| Com1(Block((l1, l3))) -> newframes(labelcom(l3), rho, sigma);
  push(labeldec(l1), top(cstack))
| _ -> failwith("no more sensible cases in commands" ) )
val itsemcl : unit -> unit = <fun>
```

L'interprete iterativo 6



```
let itsemdecl () =
  let tempstack = top(tempvalstack) in
  let continuation = top(cstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  let dl = (match top(continuation) with
    | Decl(d11) -> d11
    | _ -> failwith("impossible in semdecl")) in
  if dl = [] then pop(continuation) else
  (let currd = List.hd dl in
  let newdl = List.tl dl in pop(continuation); push(Decl(newdl), continuation);
  (match currd with

    | Decl( (i,e) ) ->
      pop(continuation);
      push(Decl(Dec2((i, e))::newdl), continuation);
      push(Exprdl(e), continuation)

    | Dec2((i,e) ) ->
      let arg = top(tempdstack) in
      pop(tempdstack);
      popenv(); pushenv(bind(rho, i, arg))

    | _ -> failwith("no more sensible cases for semdecl")))
val itsemdecl : unit -> unit = <fun>
```

L'interprete iterativo 7



```
let initState() = svuota(continuation); svuota(tempstack); svuota(tempdvalstack);
svuotaenv(); svuotastore(); svuota(labelstack)
val initState : unit -> unit = <fun>

let loop () =
  while not(empty(cstack)) do
    while not(empty(top(cstack))) do
      let currconstr = top(top(cstack)) in
      (match currconstr with
       | Expr1(e) -> itsem()
       | Expr2(e) -> itsem()
       | Exprd1(e) -> itsemden()
       | Exprd2(e) -> itsemden()
       | Coml(cl) -> itsemcl()
       | Decl(l) -> itsemdecl()
       | _ -> failwith("non legal construct in loop"))
    done;
    (match top(labelstack) with
     | Expr1(_) -> let valore = top(top(tempvalstack)) in
       pop(top(tempvalstack)); pop(tempvalstack); push(valore,top(tempvalstack));
       popenv(); popstore(); pop(tempdvalstack)
     | Exprd1(_) -> let valore = top(top(tempdvalstack)) in
       pop(top(tempdvalstack)); pop(tempdvalstack); push(valore,top(tempdvalstack));
       popenv(); popstore(); pop(tempvalstack)
     | Decl(_) -> pop(tempvalstack); pop(tempdvalstack)
     | Coml(_) -> let st = topstore() in popenv(); popstore(); popstore(); pushstore(st);
       pop(tempvalstack); pop(tempdvalstack)
     | _ -> failwith("non legal label in loop"));
    pop(cstack); pop(labelstack)
  done
val loop : unit -> unit = <fun>
```

L'interprete iterativo 8



```
let sem (e,(r: dval env), (s: mval store)) = initState();
push(emptystack(tframesize(e),Novalue),tempvalstack);
newframes(Expr1(e), r, s);
loop();
let valore= top(top(tempvalstack)) in
pop(tempvalstack); valore
val sem : exp * dval Funenv.env * mval Funstore.store -> eval = <fun>

let semden (e,(r: dval env), (s: mval store)) = initState();
push(emptystack(tframesize(e),Unbound),tempdvalstack);
newframes(Exprd1(e), r, s);
loop();
let valore= top(top(tempdvalstack)) in
pop(tempdvalstack);
valore
val semden : exp * dval Funenv.env * mval Funstore.store -> dval = <fun>

let semcl (cl,(r: dval env), (s: mval store)) = initState();
pushstore(emptystore(Undefined));
newframes(labelcom(cl), r, s);
loop();
let st = topstore() in popstore();
st
val semcl : com list * dval Funenv.env * mval Funstore.store -> mval Funstore.store = <fun>
```

L'interprete iterativo 9



```
let semdv(dl, r, s) = initState();
  newframes(labeldec(dl), r, s);
  loop();
  let st = topstore() in popstore();
  let rt = topenv() in popenv();
  (rt, st)
val semdv : (ide * exp) list * dval Funenv.env * mval Funstore.store ->
  dval Funenv.env * mval Funstore.store = <fun>

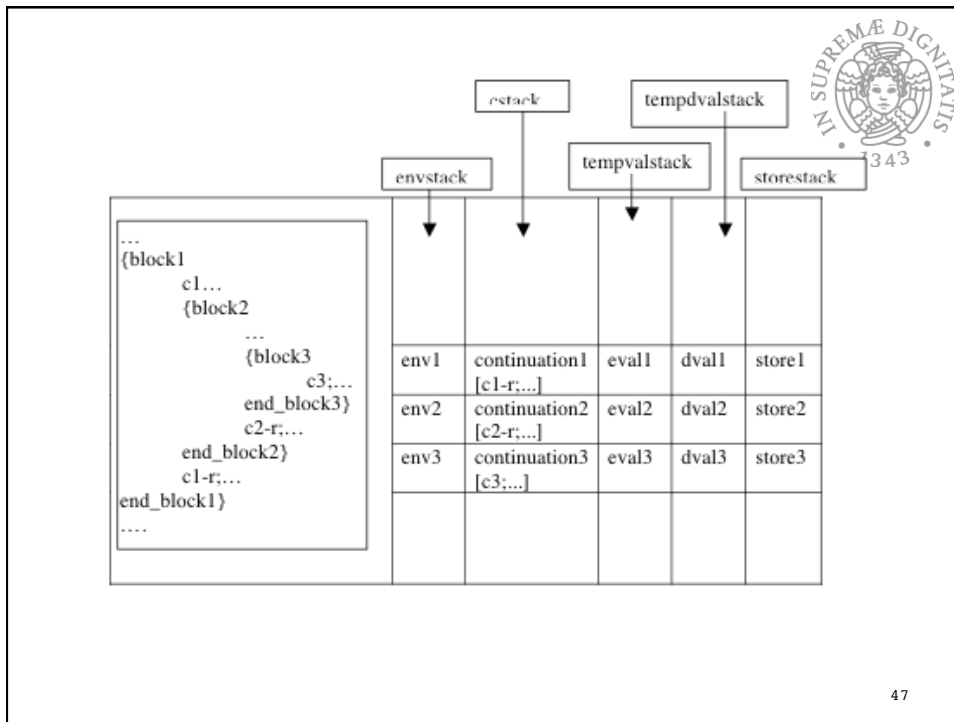
let semc((c: com), (r:dval env), (s: mval store)) = initState();
  pushstore(emptystore(Undefined));
  newframes(labelcom([c], r, s);
  loop();
  let st = topstore() in popstore();
  st
val semc : com * dval Funenv.env * mval Funstore.store -> mval Funstore.store = <fun>

let semb((dl, cl), r, s) = initState();
  pushstore(emptystore(Undefined));
  newframes(labelcom(cl), r, s);
  push(labeldec(dl), top(cstack));
  loop();
  let st = topstore() in popstore();
  st
val semb : ((ide * exp) list * com list) * dval Funenv.env * mval Funstore.store ->
  mval Funstore.store = <fun>
```

Cosa abbiamo imparato?



- 🦋 **Natura dinamica dei record di attivazione**
 - Ogni volta che si entra in un blocco si crea un record di attivazione
 - Il record di attivazione viene “eliminato” quando si esce dal blocco
 - Cosa contiene il record di attivazione
 - ✓ Ambiente
 - ✓ Spazio per la valutazione di valori intermedie
 - ✓ Informazioni di controllo



ambiente locale statico

- ✎ ambiente locale ad un blocco
 - attivato all'ingresso del blocco
 - disattivato all'uscita dal blocco
- ✎ l'associazione è utilizzabile solo quando è attiva
- ✎ le dichiarazioni vengono eseguite una sola volta
 - compilazione, caricamento, o prima esecuzione del blocco
- ✎ se il blocco contiene dichiarazioni di variabile
 - anche la memoria locale al blocco viene preservata tra due diverse esecuzioni del blocco
 - costituisce una sorta di stato interno al blocco

Ambiente locale statico: quali linguaggi?



FORTRAN

- non ci sono blocchi, ma questa è la regola per l'ambiente locale dei sottoprogrammi
- le diverse attivazioni (chiamate) dello stesso sottoprogramma condividono ambiente e memoria locali
 - ✓ la chiamata n -esima trova lo stato lasciato dalla chiamata $(n-1)$ -esima

ALGOL, PL/I, C

- alcune dichiarazioni locali (a blocchi o sottoprogrammi) possono essere dichiarate statiche (static, own, ...)
- come sopra, per queste particolari associazioni
 - ✓ le altre sono trattate con l'ambiente dinamico

Java

- le dichiarazioni static all'interno di una classe provocano la creazione di ambiente (ed eventuale memoria) che appartengono alla classe e non agli oggetti sue istanze
 - ✓ sono eseguite una sola volta all'atto della esecuzione della dichiarazione di classe
 - ✓ possono costituire uno stato interno comune a tutti gli oggetti della classe

Ambiente locale statico: perché?



una volta si argomentava

- le dichiarazioni statiche sono più efficienti (vedi dopo!)

oggi diciamo

- creano comportamenti molto più complessi da comprendere dal punto di vista semantico
 - ✓ la presenza di uno stato interno fa sì che la semantica debba tener conto di tutta la sequenza di attivazioni
- sono utili se si vogliono definire blocchi (o meglio sottoprogrammi) dotati di stato interno
 - ✓ generatore di numeri casuali, generatore di nomi sempre nuovi, etc.

in Java

- l'utilizzazione dell'ambiente locale statico per definire metodi stand-alone con il meccanismo delle classi è una forzatura legata alla visione uniforme (e manichea) di Java
 - ✓ in generale, per i sottoprogrammi, essere trattati in modo statico o dinamico non fa differenza
- le variabili statiche andrebbero usate solo quando si vuole realmente creare uno stato interno (magari condiviso)

Ambiente locale statico: cenni all'implementazione



- ✦ non abbiamo ancora visto la vera implementazione dell'ambiente locale dinamico
 - per ora abbiamo una pila di record di attivazione, in cui ogni record contiene l'intero ambiente e l'intera memoria, implementati come funzioni
 - nell'implementazione che vedremo, il record di attivazione conterrà delle tabelle che realizzano le associazioni locali (e la relativa memoria)
- ✦ con l'ambiente locale statico
 - e per le associazioni statiche in linguaggi che le permettono non è necessario inserire nulla nel record di attivazione
 - ambiente e memoria locali sono creati staticamente (dal compilatore)
 - ✓ e ci può essere un grande spreco di memoria!
 - in una tabella associata al codice (del sottoprogramma, della classe)
 - questo permette anche di eliminare
 - ✓ i nomi dalle tabelleed i riferimenti ai nomi dal codice
 - ✓ rimpiazzandoli con displacements (indirizzi relativi nelle tabelle)

Esercizio



- ✦ Consideriamo il linguaggio funzionale didattico con le dichiarazioni (costrutto let)
- ✦ Invece di usare gli identificatori potremmo usare indici per rappresentare la posizione nell'ambiente dell'identificatore

Esempio



- ✎ `Let("z", Eint 7, sum(Den "z", Den "z"))`
- ✎ `Let(Eint 7, sum(Den 0, Den 0))`
- ✎ Idea= indice e' il numero dei let "attraversati"
- ✎ `Let("z", Eint 7,
 Let("y", Eint 5, sum(Den "z", Den "y"))
)`
- ✎ `Let("z", Eint 7,
 Let("y", Eint 5, sum(Den 0, Den 1))
)`

Esercizio



- ✎ Compilare il linguaggio didattico nel linguaggio oggetto che non contiene variabili ma indici e scrivere l'interprete del linguaggio oggetto
- ✎ Modificare la sintassi astratta Texpr
 - Tden of int
 - Tlet of Texpr * texpr