AA 2014-2015

21. Controllo di sequenza: espressioni e comandi

Espressioni in sintassi astratta

- Alberi etichettati
  - nodi
    - ✓applicazioni di funzioni (operazioni primitive)
    - ✓i cui operandi sono i sottoalberi
  - Foglie
    - ✓costanti o variabili (riferimenti a dati)
- Consideriamo solo espressioni pure, che non contengono
  - definizioni di funzione
  - applicazioni di funzione
  - introduzione di nuovi nomi (blocco)
- L'unico problema semantico interessante che riguarda la valutazione di espressioni pure è quello della regola di valutazione

Operazioni primitive

- Le operazioni primitive sono in generale **funzioni parziali**
  - indefinite per alcuni valori degli input
    - ✓ errori “hardware”: overflow, divisione per zero
    - ✓ errori rilevati dal supporto a run time: errori di tipo a run time, accessi errati ad array, accessi a variabili non inizializzate, esaurimento memoria libera
  - nei linguaggi moderni tutti questi casi provocano il sollevamento di una eccezione, che può essere catturata ed eventualmente gestita
- Alcune operazioni primitive sono **funzioni non strette**
  - una funzione è non stretta sul suo i-esimo operando se ha un valore definito quando viene applicata a una n-upla di valori, di cui l’i-esimo valore è indefinito

Regole di valutazione

- **Regola interna**
  - prima di applicare l'operatore, si valutano le sottoespressioni
- **Regola esterna**
  - è l'operatore che richiede la valutazione delle sottoespressioni
- Le due regole di valutazione possono dare semantiche diverse
  - se qualcuna delle sottoespressioni ha valore "indefinito"
    - ✓errore, non terminazione, sollevamento di una eccezione, ...
  - e l'operatore è non stretto in quell'argomento, la regola esterna può calcolare comuque un valore, la regola interna no
- Esempi di tipiche operazioni primitive non strette
  - condizionale: **if true then C1 else C2**
  - operatori logici: **true or E**

Operazioni non strette: **ifthenelse**


```
if x = 0 then y else y/x
```

- In sintassi astratta

```
ifthenelse(=(x,0), y, /(y,x))
```

- Usando la **regola interna**, valuto tutti e tre gli operandi
  - se x vale 0, la valutazione del terzo operando causa un errore
    - ✓ l'intera espressione ha valore indefinito
- Usando la **regola esterna**, valuto solo il primo operando
  - se **x** vale **0**, valuto il secondo operando
  - il terzo operando non viene valutato e l'intera espressione ha un valore definito

Operazioni non strette: `or`

`true or "expr1"`

- In sintassi astratta

  or(true, "expr1")

- Usando la **regola interna**, valuto tutti e due gli operandi
  - se la valutazione del secondo operando da origine a un errore, l'intera espressione ha valore indefinito
  - in ogni caso, la valutazione di **expr1** è inutile!
- Usando la **regola esterna**, valuto il primo operando
  - se questo vale **true**, non devo fare altro, e il risultato è **true** qualunque sia il valore (anche indefinito) di **expr1**
  - altrimente viene valutato **expr1**

Regola esterna vs. interna

- La **regola esterna**
  - è sempre corretta
  - è più complessa da implementare, perché ogni operazione deve avere la propria "politica"
  - è necessaria in pochi casi, per le operazioni primitive
    - ✓ sono poche le operazioni primitive non strette
- La **regola interna**
  - non è in generale corretta per le operazioni non strette
  - è banale da implementare
- La soluzione più ragionevole
  - regola interna per la maggior parte delle operazioni
  - regola esterna per le poche primitive non strette

Paradigma funzionale

Funzionale

- Paradigma di programmazione in cui il flusso di esecuzione procede mediante la riscrittura di valori ("value oriented programming"), applicando funzioni matematiche
- Vantaggi
  - mancanza di effetti collaterali (side effect) delle funzioni, fatto che comporta una più facile verifica della correttezza
  - ottimizzazione nella programmazione parallela
    - ✓esempio: Google MapReduce framework

Paradigma funzionale

- Esaminiamo un frammento di un linguaggio funzionale “puro” e deriviamo l’interprete del linguaggio a partire dalla semantica operazionale del linguaggio

Frammento funzionale: sintassi


```
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
```

Frammento funzionale: verso l'implementazione

- Presenteremo un interprete per valutare espressioni di tipo **exp**. Si noti che ci sono i *nomi* (o *identificatori, variabili*) (**Den(i)**) ma non un costrutto **let** per creare binding, quindi per valutare una **exp** dobbiamo fornire un ambiente
- Un nome può essere legato nell'ambiente solo a un **valore esprimibile**, non a una qualunque espressione, quindi i valori ***esprimibili*** coincidono con quelli **denotabili**
- Per valutare le espressioni abbiamo bisogno di un semplice type checking dinamico. Per esempio
  - in **Prod(a,b)**, **a** ed **b** devono essere **int**, non **bool**
  - in **Ifthenelse(a,b,c)**, **a** deve essere **bool**, **b** e **c** devono avere stesso tipo

Dominio dei valori esprimibili

Definiamo i **valori esprimibili**, ovvero i valori che posso restituire come risultato della valutazione di una espressione, e che possono essere associati a variabili

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

Come mai **Unbound**? Posso avere un identificatore che non è associato ad alcun valore...

Ambiente

- Necessario per gestire associazioni del tipo **nome -> valore**
- Useremo l'ambiente polimorfo già visto

```
module Funenv:ENV =
  struct
  type 't env = string -> 't

  let emptyenv(x) = function y -> x
  let applyenv(x,y) = x y
  let bind((r: 'a env), (l: string), (e: 'a)) =
    function lu -> if lu = l then e else applyenv(r, lu)
  ...
end
```

Type checking

- Controlla se il valore esprimibile **y** è di tipo **"int"** o **"bool"**

```
let typecheck (x, y) = match x with
      | "int" ->
        (match y with
            |  Int(u) -> true
            | _ -> false)
      | "bool" ->
        (match y with
            |  Bool(u) -> true
            | _ -> false)
      | _ -> failwith ("not a valid type")

val typecheck : string * eval -> bool = <fun>
```

Semantica dei singoli operatori

- Partiamo dalle regole (ovvie) che definiscono la semantica operazionale big-step di ogni operatore
- Esempio: regole dell'if-then-else (val. non stretta)

$$\frac{env \triangleright g \Rightarrow true \quad env \triangleright e1 \Rightarrow v1}{env \triangleright ifthenelse(g,e1,e2) \Rightarrow v1}$$

$$\frac{env \triangleright g \Rightarrow false \quad env \triangleright e2 \Rightarrow v2}{env \triangleright ifthenelse(g,e1,e2) \Rightarrow v2}$$

Implementazione operazioni primitive

- Una funzione per ogni operazione primitiva: controlla i tipi e calcola il risultato

```
let minus x = if typecheck("int", x)
        then (match x with Int(y) -> Int(-y))
        else  failwith ("type error")

let iszero x = if typecheck("int", x)
        then(match x with Int(y) -> Bool(y = 0))
        else  failwith ("type error")

let equ(x,y) = if typecheck("int”, x) & typecheck("int", y)
     then (match (x,y) with (Int(u), Int(w)) -> Bool(u = w))
     else failwith ("type error")

let plus(x,y) = if typecheck("int", x) & typecheck("int", y)
     then  (match (x,y) with (Int(u), Int(w)) -> Int(u + w))
     else failwith ("type error")
```

La semantica operazionale


```
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")

val sem : exp * eval Funenv.env -> eval = <fun>
```

Operatori stretti o no?

And e Or interpretati come funzioni strette

```
...
  | And(a,b) ->  et(sem(a, r), sem(b, r))
  | Or(a,b)  ->  vel(sem(a, r), sem(b, r))
```

Condizionale interpretato (ovviamente!) come funzione non stretta, sfruttando if-then-else di OCaml

```
...
  | 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")
```

La semantica operazionale è un interprete


```
val sem : exp * eval Funenv.env -> eval = <fun>
```

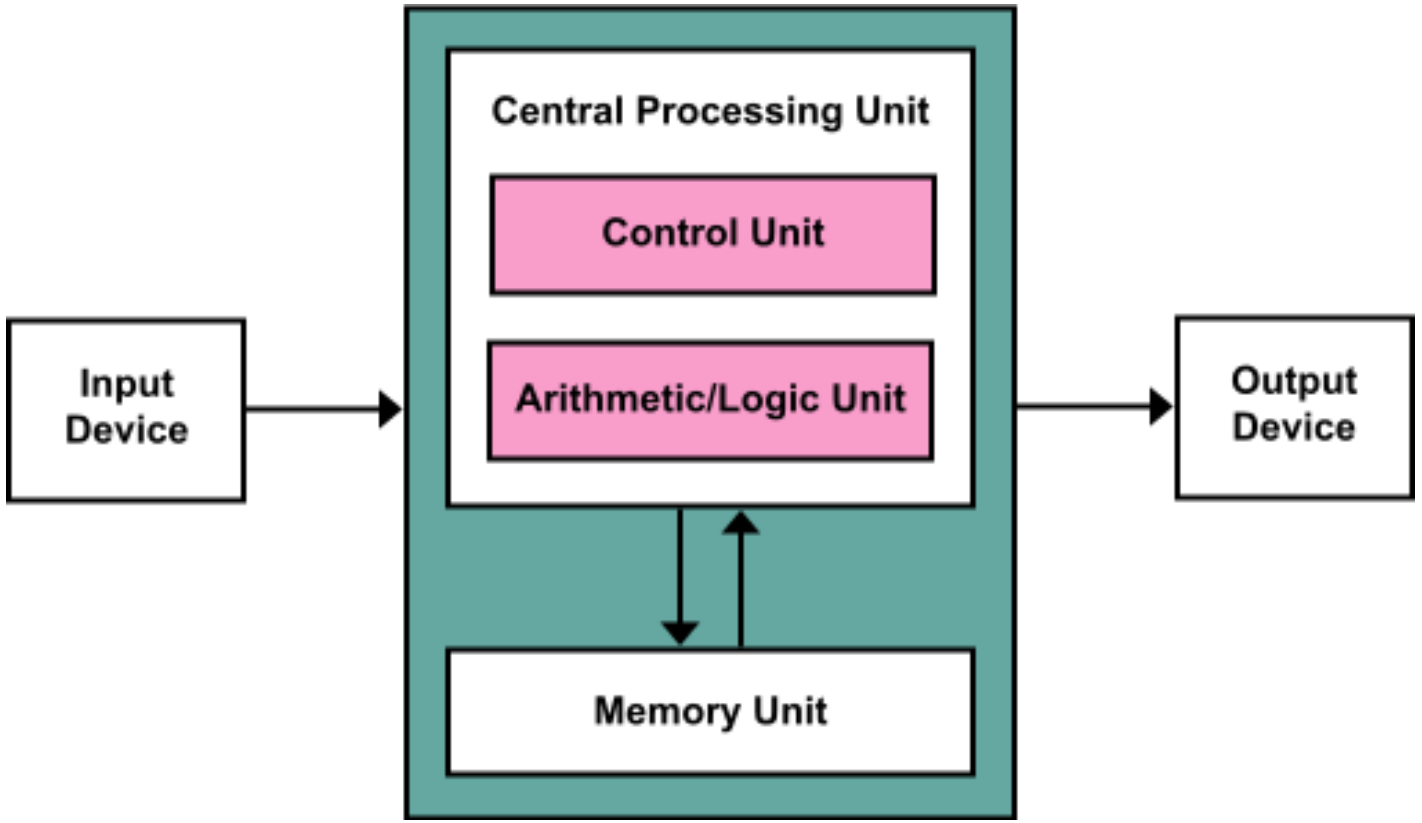
- Definito in modo ricorsivo: utilizzando la ricorsione del metalinguaggio (linguaggio di implementazione)
- Avevamo visto come si poteva eleminare la ricorsione dall'interprete
  - introducendo **continuation stack** e **stack di valori temporanei**
    - ✓ si ottiene una versione di livello più basso
    - ✓ più vicina ad una "vera" implementazione in termini di una macchina virtuale a stack

Paradigma imperativo

Paradigma imperativo

- Un programma viene inteso come un insieme di istruzioni che trasformano lo stato della macchina virtuale
- La computazione procede modificando valori che sono memorizzati in opportune locazioni della memoria
- Modello derivato dalla **Macchina di von Neumann**, prototipo dei moderni elaboratori

Macchina di van Neumann



Macchina a programma memorizzato: i dati e le istruzioni del programma risiedono nello stesso spazio di memoria

Effetti collaterali, comandi, espressioni pure

- Assumiamo che continuino ad esistere le espressioni
  - diverse dai comandi perché la loro semantica non modifica la memoria (non produce effetti laterali) e restituisce un valore (un **eval**)
- Questo approccio non è quello di C
  - nel quale quasi ogni costrutto può restituire un valore e modificare lo stato

Comandi e espressioni

- La distinzione (semantica) tra espressioni e comandi è difficile da mantenere se si permette che i comandi possano occorrere all'interno di espressioni, soprattutto in presenza di "operazioni definite dal programmatore" (funzioni)
- Nel linguaggio didattico forzeremo tale distinzione
  - permettendo "effetti collaterali" solo in alcuni costrutti che avranno una semantica diversa

Un frammento di linguaggio imperativo: domini sintattici


```
type ide = string
type exp =
        | Eint of int
        | Ebool of bool
        | Den of ide    (* valore associato a identificatore *)
        | 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    (* valore di una variabile *)

type com =
        | Assign of exp * exp
        | Cifthenelse of exp * com list * com list
        | While of exp * com list
```

Domini semantici

- Per caratterizzare lo stato di esecuzione del programma serve, oltre all'ambiente, la memoria
- Ai domini semantici dei valori si aggiungono le **locazioni**
  - che decidiamo non essere né esprimibili né memorizzabili
- Tre domini distinti di valori
  - **eval -** valori esprimibili, possibili risultati di espressioni
  - **dval** - valori denotabili, possono essere associati a variabili nell'ambiente
  - **mval -** valori memorizzabili, possono essere associati a locazioni nella memoria
- Con operazioni di "conversione" (**casting**)
- Introduciamo una funzione di valutazione semantica (**semden**) che calcola un **dval** invece che un **eval**

La memoria: specifica


```
module type STORE =
    sig
      type 't store
      type loc
      val emptystore : 't -> 't store
      val allocate : 't store * 't -> loc * 't store
      val update : 't store * loc * 't -> 't store
      val applystore : 't store * loc -> 't
    end
```

La memoria: implementazione


```
module Funstore:STORE =
   struct
     type loc = int

     type 't store = loc -> 't

     let (newloc, initloc) = let count = ref(-1) in
          (fun ( ) -> count := !count +1; !count),
          (fun ( ) -> count := -1)

     let emptystore(x) = initloc( ); function y -> x

     let applystore(x,y) = x y

     let allocate((r: 'a store), (e: 'a)) =
       let l = newloc( ) in
          (l, function lu -> if lu = l then e
                                       else applystore(r, lu))

     let update((r: 'a store),(l: loc), (e: 'a)) =
          function lu -> if lu = l then e else applystore(r, lu)
   end
```

I domini dei valori


```
exception Nonstorable
exception Nonexpressible

(* valori esprimibili, possibili risultati di espressioni *)
type eval =
        | Int of int
        | Bool of bool
        | Novalue

(* valori denotabili nell'ambiente *)
type dval =
        | Dint of int
        | Dbool of bool
        | Unbound
        | Dloc of loc

(* valori memorizzabili *)
type mval =
        | Mint of int
        | Mbool of bool
        | Undefined
```

Casting tra vari tipi di valori


```
let evaltomval e = match e with
                 | Int n -> Mint n
                 | Bool n -> Mbool n
                 | _ -> raise Nonstorable

let mvaltoeval m = match m with
                 | Mint n -> Int n
                 | Mbool n -> Bool n
                 | _ -> Novalue

let evaltodval e = match e with
                 | Int n -> Dint n
                 | Bool n -> Dbool n
                 | Novalue -> Unbound

let dvaltoeval e = match e with
                 | Dint n -> Int n
                 | Dbool n -> Bool n
                 | Dloc n -> raise Nonexpressible
                 | Unbound -> Novalue
```

Semantica delle espressioni


```
let rec sem ((e: exp), (r: dval env), (s: mval store)) =
      match e with
       | Eint(n) -> Int(n)
       | Ebool(b) -> Bool(b)
(* semantica di una variabile i: il valore associato
   nell'ambiente env; eccezione se è una loc *)
       | 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))
<continua>
```

Semantica delle espressioni


```
<continua>
      | 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")
(* se e è un identificatore associato a una locazione, ne
   restituisce il valore, altrimenti fallisce *)
      | Val(e) -> match semden(e, r, s) with
            | Dloc(n) -> mvaltoeval(applystore(s, n))
            | _ -> failwith("not a variable")

let semden ((e: exp), (r: dval env), (s: mval store)) =
  match e with
      | Den(i) -> applyenv(r, i)
      | _ -> evaltodval(sem(e, r, s))

val sem : exp * dval Funenv.env * mval Funstore.store -> eval = <fun>

val semden : exp * dval Funenv.env * mval Funstore.store -> dval = <fun>
```

Semantica dell'assegnamento

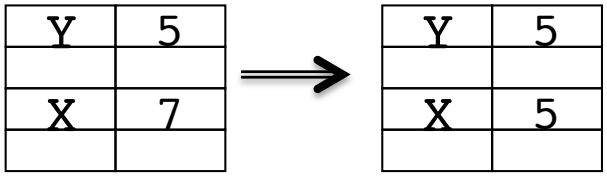
- Sintassi
  - sintassi astratta — Pascal — C, Java
  - **`Assign(e1,e2)`** — **`e1 := e2`** — **`e1 = e2`**
- Significato
  - si valuta **e1** e si deve ottenere una locazione **L** (un "L-value")
  - si valuta **e2** e si deve ottenere un valore memorizzabile **V**
  - si cambia la memoria associando alla locazione **L** il valore **V**
- NB.
  - **e1** può essere una variabile, un elemento di array, un campo di record, … In generale può contenere sottoespressioni con effetti collaterali (ad esempio: **a[i++]**)
  - quindi l'ordine in cui si valutano **e1** ed **e2** è importante

Assegnamento tra variabili

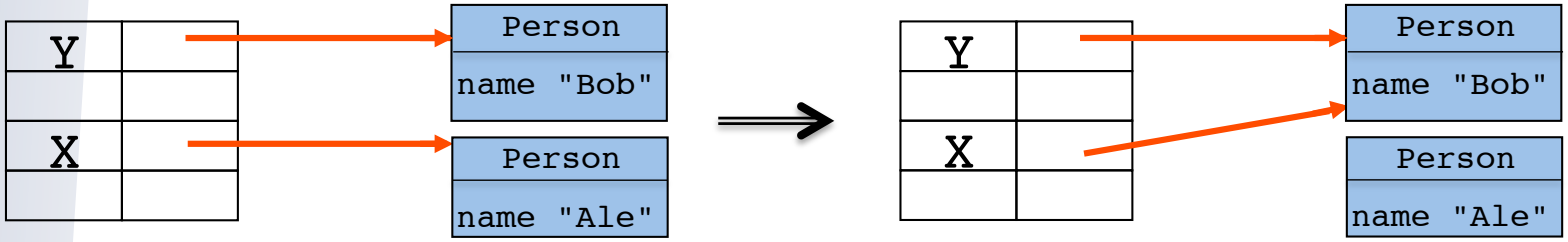
- Ne consegue che nel caso di assegnamento tra variabili, ad es.

X := Y

- il ruolo delle due variabili è diverso: di **X** si considera la locazione (L-value) mentre di **Y** il valore nella locazione associata (R-value)
- Effetto: copia il contenuto della locazione di **Y** nella locazione di **X**
- Per valori primitivi, questo crea una copia



- Per strutture dati allocate nello heap, l'assegnamento crea la copia di un puntatore, causando lo sharing della struttura



Assegnamento: regola di valutazione

$$\frac{env, store \triangleright e1 \Rightarrow v1 \qquad v1 = Dloc(n) \qquad env, store \triangleright e2 \Rightarrow v2 \qquad \mathrm{v} = \mathrm{evaltomval(v2)}}{env, store \triangleright assign(e1, e2) \Rightarrow store[v / n]}$$

While: regole di valutazione

$$\frac{env, store \triangleright g \Rightarrow false}{env, store \triangleright while(g, cl) \Rightarrow store}$$

$$\frac{env, store \triangleright g \Rightarrow true \quad env, store \triangleright cl; while(g,cl) \Rightarrow store'}{env, store \triangleright while(g, cl) \Rightarrow store'}$$

Semantica operazionale: comandi


```
let rec semc((c: com), (r: dval env), (s: mval store)) = match c with
        | Assign(e1, e2) ->
           (match semden(e1, r, s) with
                 | Dloc(n) -> update(s, n, evaltomval(sem(e2, r, s)))
                 | _ -> failwith ("wrong location in assignment"))
        | Cifthenelse(e, cl1, cl2) -> let g = sem(e, r, s) in
              if typecheck("bool", g) then
               (if g = Bool(true) then semcl(cl1, r, s)
                                  else semcl(cl2, r, s))
              else failwith ("nonboolean guard”)
        | While(e, cl) -> let g = sem(e, r, s) in
             if typecheck("bool", g) then
                (if g = Bool(true) then
                            semcl((cl @ [While(e, cl)]), r, s)
                                   else s)
           else failwith ("nonboolean guard")
and semcl((cl: com list), (r: dval env), (s: mval store)) = match cl with
        | [ ] -> s
        | c::cl1 -> semcl(cl1, r, semc(c, r, s))
```

val **semc**:

com * dval Funenv.env * mval Funstore.store

-> mval Funstore.store = <fun>

val **semcl**:

com list * dval Funenv.env * mval Funstore.store

-> mval Funstore.store = <fun>

Eliminare la ricorsione

- Per le espressioni, bisogna prevedere il caso in cui il valore è un **dval**
  - nuova pila di valori denotabili temporanei
  - diverse etichette per le espressioni
- Per i comandi, la ricorsione può essere rimpiazzata con l'iterazione senza utilizzare ulteriori pile
- Il dominio dei comandi è "quasi" **tail-recursive**, dato che non è mai necessario valutare i due rami del condizionale
  - si può utilizzare la struttura sintattica (lista di comandi) per mantenere l'informazione su quello che si deve ancora valutare
    - ✓ basta una unica cella
    - ✓ che possiamo "integrare" nella pila di espressioni etichettate
- Il valore restituito dalla funzione di valutazione semantica dei comandi (uno store!) può essere gestito come aggiornamento di una "variabile globale" di tipo store