PROGRAMMAZIONE 2

22. Espressioni:

INTERPRETE in OCAML

Espressioni logiche: sintassi astratta

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

Interprete di espressioni logiche (True, False, And, Not)

```
let rec eval exp =
     match exp with
       True -> True
     | False -> False
     | Not(exp0) -> match eval exp0 with
                      True -> False
                    | False -> True
     | And(exp0,exp1) ->
           match (eval exp0, eval exp1) with
             (True,True) -> True
           | (_,False) -> False
           | (False,_) -> False
```

Valutazione AND

REGOLA LOGICA

$$\frac{e_1 \Rightarrow v_1, e_2 \Rightarrow v_2}{e_1 \ \mathbf{and} \ e_2 \Rightarrow v_1 \wedge v_2}$$

Tabella: $true \wedge true = true$ ... .

INTERPRETE

```
And(exp0,exp1) ->
            match (eval exp0, eval exp1) with
                (True,True) -> True
              | (_,False) -> False
              | (False,_) -> False
```

Espressioni a valori interi : sintassi astratta

```
type exp = …
   | CstI of int
   | Sum of  exp * exp
   | Times of exp * exp
```

Costanti e operazioni binarie standard

Valutazione delle espressioni

```
let rec eval exp =
      match exp with
       ……
      | CstI(i) -> i
      | Sum(e1,e2)   -> eval e1 + eval e2
      | Times(e1,e2)   -> eval e1 * eval e2
```

Dichiarazioni: sintassi astratta

```
type exp = ….
   | Var of string
   | Let of string * exp * exp
```

Per esempio

```
Let("z", CstI 17, Sum(Var "z", Var "z"))
```

In sintassi concreta

```
let z = 17 in z + z
```

Ambiente

- Per definire l'interprete dobbiamo introdurre una struttura di implementazione (run-time structure) che permetta di recuperare i valori associati agli identificatori
- La valuzione delle espressioni diventa parametrica rispetto all' ambiente
- L' ambiente viene modificato all'entrata e all'uscita di un blocco

Ambiente: implementazione (naïve)

```
let emptyenv = []
(* the empty environment *)

let rec lookup env x =
    match env with
    | []          -> failwith ("not found")
    | (y, v)::r -> if x = y then v else lookup r x
```

Regole di valutazione e interprete

$$\frac{env(x) = v}{env \triangleright Var\ \mathrm{x} \Rightarrow v}$$

```
eval (Var x) env -> lookup env x
```

$$\frac{env \rhd e_1 \Rightarrow v_1 , env \rhd e_2 \Rightarrow v_2}{env \rhd Sum(e_1, e_2) \Rightarrow v_1 + v_2}$$

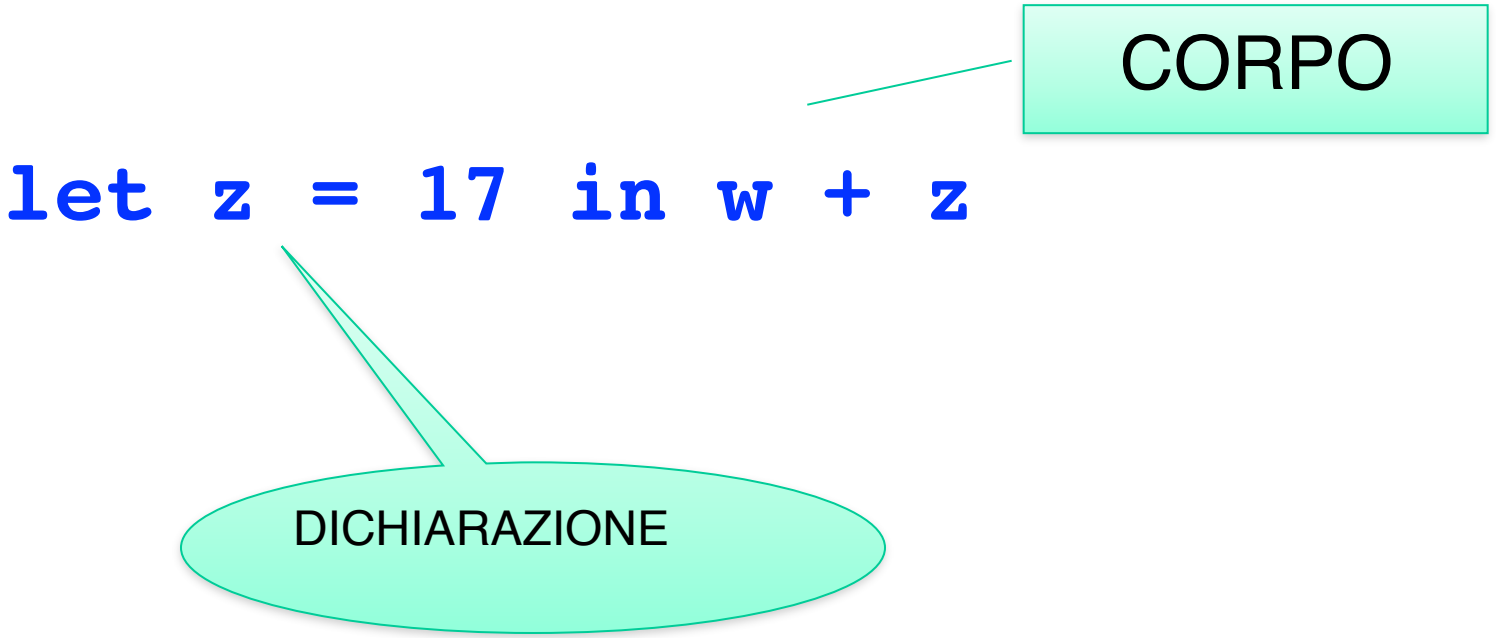
```
eval Sum(e1, e2) env -> eval e1 env + eval e2 env
```

Interprete: semplici espressioni intere

(* la valutazione è parametrica rispetto a **env** *)

```
let rec eval e (env : (string * int) list) : int =
    match e with
     | CstI i                -> i
     | Var x                 -> lookup env x
     | Sum(e1, e2) -> eval e1 env + eval e2 env
     | Times(e1, e2) -> eval e1 env * eval e2 env
     | Minus(e1, e2) -> eval e1 env - eval e2 env
     |  _                ->  failwith "unknown primitive"
```

Aggiungiamo le dichiarazioni



Regola per il costrutto **`Let`**

$$\frac{env \triangleright erhs \Rightarrow xval \quad env[xval / x] \triangleright ebody \Rightarrow v}{env \triangleright \text{Let x = erhs in ebody} \Rightarrow v}$$

```
eval (Let(x, erhs, ebody)) env ->
      let xval = eval erhs env in
        let env1 = (x, xval) :: env in
          eval ebody env1
```

- Si valuta **`erhs`** nell'ambiente corrente ottenendo **`xval`**
- Si valuta **`ebody`** nell'ambiente esteso con il legame tra **`x`** e **`xval`** *(se esiste già lo copre)* ottenendo il valore **`v`**
- La valutazione del **`let`** nell'ambiente corrente produce il valore **`v`**

Interprete: espressioni intere con dichiarazioni

```
let rec eval e (env : (string * int) list) : int =
    match e with
      | CstI i              -> i
      | Var x               -> lookup env x
      | Sum(e1, e2) -> eval e1 env + eval e2 env
      | Times(e1, e2) -> eval e1 env * eval e2 env
      | Minus(e1, e2) -> eval e1 env - eval e2 env
      | Let(x, erhs, ebody) ->
          let xval = eval erhs env in
             let env1 = (x, xval) :: env in
                 eval ebody env1
    |  _    ->  -> failwith "unknown primitive"
```

Condizionale

- Sintassi concreta
  - `e::=···e1 == e2 |if e then e1 else e2`
- Sintassi astratta
  - ```
    type exp = ….
         | Eq of exp * exp
         | Cond of exp * exp * exp
    ```
    ```

Valutazione: uguaglianza

REGOLA LOGICA

$$\frac{e_1 \Rightarrow v, e_2 \Rightarrow v}{Eq(e_1, e_2) \Rightarrow True}$$

INTERPRETE

```
Eq(exp0,exp1) ->
      match (eval exp0 env, eval exp1 env)
                  with
                        (n, n) -> True
                      | (-,-) -> False
```

Valutazione: condizionale

REGOLA LOGICA

$$\frac{env \triangleright e \Rightarrow true, e_1 \Rightarrow v}{env \triangleright Cond(e, e_1, e_2) \Rightarrow v} \qquad \frac{env \triangleright e \Rightarrow false, e_2 \Rightarrow v'}{env \triangleright Cond(e, e_1, e_2) \Rightarrow v'}$$

INTERPRETE

```
eval Cond(g,exp0,exp1) env ->
  match eval g env with
        True -> eval exp0 env
      | False -> eval exp1 env
      | _  -> failwith("non Boolean guard")
```

Come si risolve il problema?

- Introdurre la nozione di tipi esprimibili
  - `type evT = Int of int | Bool of bool`
- Modificare l'ambiente e le regole dell'interprete di conseguenza
- Esempio

```
let rec eval e (env : (string * evT) list) : evT
```

Basta?

- Potremmo scrivere espressioni strane come questa
  - `e1 = 1+(2==3)`
- Oppure come questa:
  - `e2 =if 1 then 2 else 3`
- Le regole di valutazione non permettono di derivare un valore per queste espressioni
- L' interprete darebbe un errore a run time

La nostra risposta: usare i tipi

- Utilizzare annotazioni di tipo a tempo di esecuzione per controllare che:
  - Argomenti delle operazioni aritmetiche e del test di uguaglianza siano valori numerici
  - La guardia di un condizionale sia una espressione booleana
- La nostra soluzione: run-time type checking

Run-Time Type Checking

```
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 * evT -> bool = <fun>
```

Operazioni elementari

- La decodifica delle operazioni elementari del linguaggio delle espressioni deve essere modificate in modo tale da tenere conto del controllo di tipi a run-time

Modifica dell'interprete

$$\frac{env \rhd e_1 \Rightarrow v_1 , env \rhd e_2 \Rightarrow v_2, v_1 : int, v_2 : int}{env \rhd Sum(e_1, e_2) \Rightarrow v_1 + v_2 : int}$$

```
let rec eval e env = match e with
  | CstInt(n) -> Int(n)
  | CstTrue -> Bool(true)
  | CstFalse -> Bool(false)
  | Sum(e1, e2) -> int_plus ((eval e1 env), (eval e2 env))
```

Condizionale

REGOLA LOGICA

$$\frac{env \triangleright e \Rightarrow true, env \triangleright e_1 \Rightarrow v}{env \triangleright Cond(e, e_1, e_2) \Rightarrow v}$$

$$\frac{env \triangleright e \Rightarrow false, env \triangleright e_2 \Rightarrow v'}{env \triangleright Cond(e, e_1, e_2) \Rightarrow v'}$$

INTERPRETE

```
let rec eval e env = match e with ….
      Cond(g,e0,e1) -> match eval g env with
     (match (typecheck("bool", g), g) with
          |(true, Bool(true)) -> eval e0 env
          |(true, Bool(false)) -> eval e0 env
          |(_, _) -> failwith ("nonboolean guard")
                )
```

Variabili libere

- In logica una variabile in una formula è libera se non compare nella portata di un quantificatore associato a tale variabile, altrimenti è legata

- Esempio: $\forall x.(P(x) \wedge Q(y))$
  [o $(Ex.\ P(x) \wedge Q(y))$ nella sintassi di LPP]
  - x è legata
  - y è libera

Occorrenze libere

- La nozione di variabile libera o legata si applica anche al caso del costrutto **`let`**
- Infatti il costrutto **`let`** si comporta come un quantificatore per la variabile che introduce
- Un identificatore **`x`** si dice "legato" se appare nel **`ebody`** dell'espressione **`let x = ehrs in ebody`**, altrimenti si dice libero
- Esempi
  - **`let z = x in z + x`** (* z legata, x libera *)
  - **`let z = 3 in let y = z + 1 in x + y`**
    (* z, y legate, x libera *)

Interpretazione di espressioni

- L'interprete introdotto ci permette di valutare espressioni costruite con la sintassi indicata
- Se una espressione ha variabili libere allora la valutazione dipende dall' ambiente