Espressioni

INTERPRETE in OCAML

Espressioni logiche: sintassi astratta

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

Definizione della sintassi astratta tramite i tipi algebrici di OCaml

Dalle regole di valutazione all'interprete OCaml

- Obiettivo: definire una funzione **eval** tale che **eval(e) = v** se e solo se **e => v**
- Esempio: dalla regola

$$\frac{e \Rightarrow v}{not\ \mathrm{e} \Rightarrow \neg v}$$

- otteniamo il seguente codice OCaml

```
eval Not(exp0) -> match eval exp0 with
                    True -> False
                  | False -> True
```

3

Valutazione AND

REGOLA LOGICA

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

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

INTERPRETE

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

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
```

5

Regole di corto circuito

- I linguaggi di programmazione (C, Java) forniscono come primitive le operazioni logiche “and” , “or” valutate con regole di **corto circuito**
- Sintassi Concreta
  - e ::=⋯|e_1 && e^2 |e_1 || e_2
- e_1 && e_2 restituisce subito il valore di verità **false** se e_1 viene valutata a **false** (la valutazione del secondo argomento è inutile).
  e_1 || e_2 restituisce subito il valore di verità **true** se e_1 viene valutata a **true** (la valutazione del secondo argomento è inutile).

Valutazione &&

REGOLA LOGICA

$$\frac{e_1 \Rightarrow false}{e_1 \,\&\&\, e_2 \Rightarrow false}$$

$$\frac{e_1 \Rightarrow true_1, e_2 \Rightarrow v_2}{e_1 \,\&\&\, e_2 \Rightarrow v_2}$$

INTERPRETE

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

Espressioni a valori interi

8

Sintassi OCaml (astratta)

```
type expr = …
  | CstI of int                    // costanti intere
  | Sum of  expr * expr            // operatori binary
  | Times of expr *expr
```


Regole di valutazione e Interprete

$$n \Rightarrow n$$

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

```
eval  CstI(n) -> n
```

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

Dichiarazioni

Espressioni con dich.: sintassi astratta

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

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

13

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
```

14

Regole di valutazione e interprete

$\triangleright$

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

eval (Var x) env -> lookup env x

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

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

15

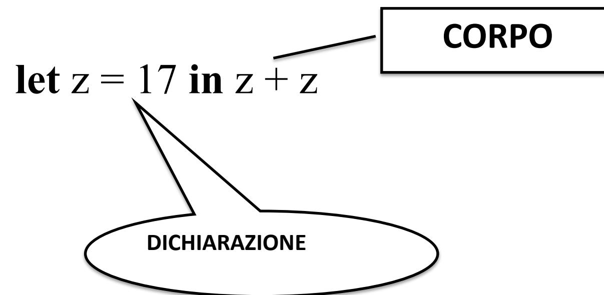
Interprete per semplici espressioni intere

(* 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
    | Prim("+", e1, e2) -> eval e1 env + eval e2 env
    | Prim("*", e1, e2) -> eval e1 env * eval e2 env
    | Prim("-", e1, e2) -> eval e1 env - eval e2 env
    | Prim _            -> failwith "unknown primitive"
```

16

Aggiungiamo le dichiarazioni



17

Regola del Let

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

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

- Si valuta **ehrs** nell'ambiente corrente ottenendo **xval**
- Si valuta **ebody** nell'ambiente esteso con il legame tra **x** e **xval** ottenendo il valore **v**
- La valutazione del "let" nell'ambiente corrente produce il valore **v**

18

Interprete per espressioni
con dichiarazioni

```
let rec eval e (env : (string * int) list) : int =
    match e with
    | CstI i            -> i
    | Var x             -> lookup env x
    | Let(x, erhs, ebody) ->
        let xval = eval erhs env in
           let env1 = (x, xval) :: env in
               eval ebody env1
    | SUM(e1, e2) -> eval e1 env + eval e2 env
    | TIMES(e1, e2) -> eval e1 env * eval e2 env
    | MINUSe1, e2) -> eval e1 env - eval e2 env
    |  _           -> failwith "unknown primitive"
```

19

Conizionale: if then else

- 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 EQ

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 eval exp1) with
              (n, n) -> True
            | (-,-) -> False
```

Valutazione Condizionale

REGOLA LOGICA

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

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

INTERPRETE

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

Per essere precisi

- Introdurre la nozione di tipi esprimibili
  - type evT = Int of int | Bool of bool
- Modificare le regole dell'interprete di conseguenza
- let rec eval (e : exp) : evT =

```
    match e with
      | CstTrue -> Bool(true)
      | CstFalse -> Bool(false)
      | CstI n -> Int(n)
    :
```

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 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
- Una prima 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

Esempio

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

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), (evale2 env))
```

Condizionale

REGOLA LOGICA

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

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

INTERPRETE

```
eval Cond(g,e0,e1) env ->
  match eval g env with
    (match (typecheck("bool", g), g) with
        |(true, Bool(true)) -> eval e0 env
        |(true, Bool(false)) -> eval e1 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: $\forall x.(P(x) \land Q(y))$
  [o (∀x. P(x) ∧ Q(y)) nella sintassi di LPP]
  - x è legata
  - y è libera

31

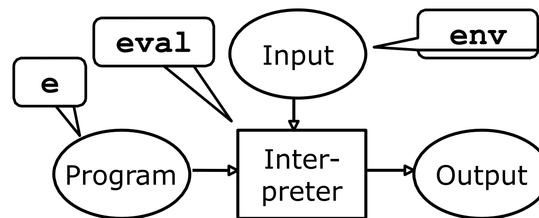
Occorrenze libere

- 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 *)

32

Interpretazione di espressioni

- L'interprete introdotto ci permette di valutare espressioni costruite con la sintassi indicata



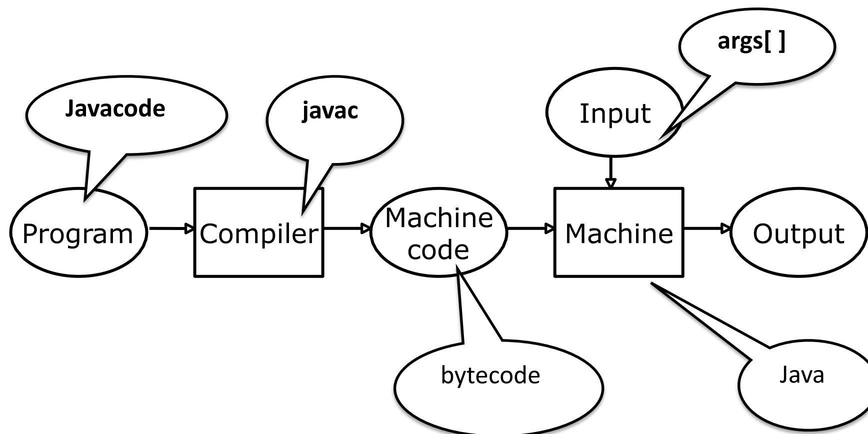
33

Verso la compilazione

- Il nostro interprete di espressioni ogni volta che deve determinare il valore associato a una variabile effettua una operazione di lookup nell'ambiente: questo potrebbe essere oneroso
- Idea: ottimizzare l'esecuzione introducendo un piccolo compilatore che traduce tutte le occorrenze di identificatori in "indici di accesso", in modo tale che l'operazione di lookup sia eseguita senza effettuare ricerche sull'ambiente, ma in modo diretto (complessità O(1))

34

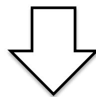
Ricordiamo lo schema misto compilazione/interpretazione



35

Le variabili: da nomi a indici

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



COMPILAZIONE

```
Let(CstI 17, Sum(Var 0, Var 0))
```

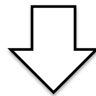
Il valore 0 indica il binding (let) più vicino

36

Indici per variabili

Idea: indice di una variabile =
numero dei **let** che si attraversano per raggiungerla

```
Let("z", CstI 17,
      Let("y", CstI 25,
             Sum(Var "z", Var "y”)))
```



COMPILAZIONE

```
Let(CstI 17,
      Let(CstI 25,
             Sum(Var 1, Var 0)))
```

37

Indici per variabili

- L'idea di utilizzare indici al posto di variabili in modo tale da avere implementazioni efficienti nasce nella teoria del lambda-calcolo con gli indici di De Bruijn

http://en.wikipedia.org/wiki/De_Bruijn_index

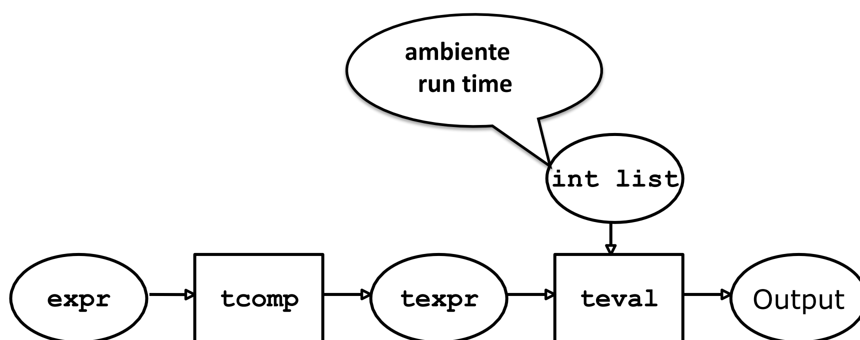
38

Il linguaggio intermedio

TARGET EXPRESSION

```
type texpr =                    (* target expression *)
  | TCstI of int
  | TVar of int                 (* indice a run time *)
  | TLet of texpr * texpr       (* erhs e ebody *)
  | TSum of texpr * texpr
```


Compilazione in codice intermedio



Compilazione in codice intermedio

```
(* Compila Expr in Texpr. Usa lista di identificatori *)

let rec tcomp e (cenv : string list) : texpr =
    match e with
      | CstI i -> TCstI i
      | Var x  -> TVar (getindex cenv x)
      | Let(x, erhs, ebody) ->
            let cenv1 = x :: cenv in
               TLet(tcomp erhs cenv, tcomp ebody cenv1)
      | Sum(e1, e2) ->
               Tsum(tcomp e1 cenv, tcomp e2 cenv)

let rec getindex cenv x =
    match cenv with
    | [] -> failwith("Variable not found")
    | y::yr -> if x=y then 0 else 1 + getindex yr x
```

41

Interpretaz. in codice intermedio

```
(* Ambiente a run time e' una lista di interi *)

open list

let rec teval (e : texpr) (renv : int list) : int =
    match e with
      | TCstI i -> i
      | TVar n  -> nth renv n
      | TLet(erhs, ebody) ->
           let xval = teval erhs renv in
              let renv1 = xval :: renv  in
                 teval ebody renv1
      | Sum(e1, e2) -> teval e1 renv + teval e2 renv
      | Times(e1, e2) -> teval e1 renv * teval e2 renv
      | _              -> failwith("unknown primitive")
```

42

Codice intermedio

- Rappresentare il programma sorgente in un codice intermedio è una tecnica che permette di dominare la complessità della implementazione di un linguaggio di programmazione
- La rappresentazione in codice intermedio permette di effettuare numerose ottimizzazioni sul codice (nel nostro caso, l'eliminazione dei nomi a run-time)
- Esempi
  - Java bytecode: codice intermedio della JVM
  - Microsoft Common Intermediate Language: codice intermedio .NET

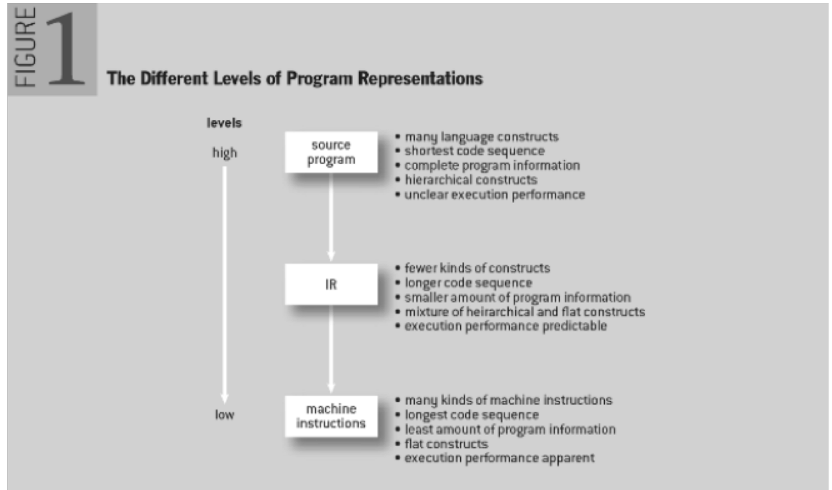
43

Cosa abbiamo imparato

- Una tecnica generale usata nei back-end dei compilatori nella fase di generazione del codice per ottenere un codice maggiormente efficiente
- Usare codice intermedio e ottimizzare il codice oggetto sul codice intermedio
- Articolo divulgativo (ma tecnico) disponibile on-line: Fred Chow, Intermediate Representation, Communications of ACM 56 (12) 2013

44

Visione Java



Visione Common Interm. Language