Realizzare un interprete in OCaml

1

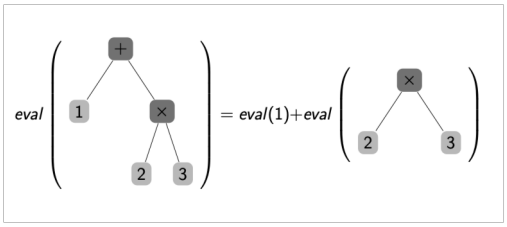
Valori e valutazione

- Consideriamo un semplice linguaggio per scrivere espressioni aritmetiche
  - $Exp\ e ::= e_1 + e_2 \mid e_1 * e_2 \mid n \in Int$
- Le constanti intere sono valori che non devono essere valutati ulteriormente
  - $Values\ \ni v ::= n\ \in Int$

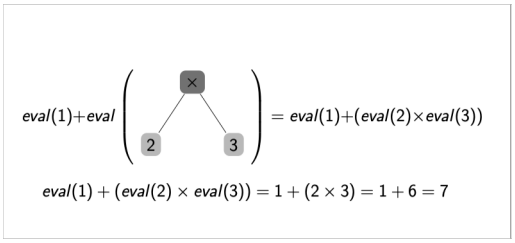
Valutazione -- eval

- Quale è il valore di una espressione *e, eval(e)*?
- *e = n*, allora *eval(e) = n* (la costante *n)*.
- $e = e_1 + e_2$, se eval(e_1) = v_1 & eval(e_1) = v_2 allora eval($e_1 + e_2$) = $v_1 + v_2$
- $e = e_1 * e_2$, se eval(e_1) = v_1 & eval(e_1) = v_2 allora eval($e_1 * e_2$) = $v_1 * v_2$

Graficamente



Graficamente (2)



OCAML

- type op = Plus | Times
- type exp =
  Int_e of int
  | Op_e of exp * op * exp

- AST -- 1 + (2 * 3)
- Op_e (Int_e(1), Plus,
  Op_e(Int_e(2),
  Times,
  Int_e(3)))

Decodifica operazioni

```
• let eval_op (v1:exp) (op:operand) (v2:exp) : exp =
    match v1, op, v2 with
        | Int_e i, Plus, Int_e j -> Int_e (i + j)
      | Int_e i, Times, Int_e j -> Int_e (i * j)
      | _ , (Plus | Times), _ ->
        if is_value v1 && is_value v2
                then raise failwith "Type_Error"
              else raise failwih "Not_Value"
• let is_value (e : exp) : bool = match e with
  | Int_e _ -> true
  | Op_e _ -> false;;
```

Interprete

```
let rec eval (e : exp) : exp =
    match e with
    | Int_e i -> Int_e i
    | Op_e(e1,op,e2) -> let v1 = eval e1 in
                                let v2 = eval e2 in
                                 eval_op v1 op v2
```

```
# let e = Op_e(Int_e(5), Times, Int_e(10));;
val e : exp = Op_e (Int_e 5, Times, Int_e 10)
# eval e;;
- : exp = Int_e 50
#
```

Regole di valutazione

Sia exp una espressione e *v* un valore, diciamo che v è il valore di exp e se valutando exp otteniamo v come risultato.

$\boxed{\text{exp} \Rightarrow v}$

Regole di valutazione

Sia exp una espressione e un valore, diciamo che v è il valore di exp e se valutando exp otteniamo v come risultato.

$\boxed{\text{exp} \Rightarrow v}$

Valutazione di valori

$\boxed{\text{v} \Rightarrow v}$

Regole di valutazione

Sia exp una espressione e un valore, diciamo che v è il valore di exp e se valutando exp otteniamo v come risultato.

$\boxed{\text{exp} \Rightarrow v}$

Valutazione di operatori

$$\boxed{\frac{\text{exp1} \Rightarrow v1\,,\text{exp2} \Rightarrow v2}{exp1 * exp2 \Rightarrow v1 * v2}}$$

$$\boxed{\frac{\text{exp1} \Rightarrow v1\,,\text{exp2} \Rightarrow v2}{exp1 + exp2 \Rightarrow v1 + v2}}$$

Regole di valutazione

$\boxed{\text{v} \Rightarrow v}$

$$\boxed{\frac{\text{exp1} \Rightarrow v1\,,\text{exp2} \Rightarrow v2}{exp1 * exp2 \Rightarrow v1 * v2}}$$

$$\boxed{\frac{\text{exp1} \Rightarrow v1\,,\text{exp2} \Rightarrow v2}{exp1 + exp2 \Rightarrow v1 + v2}}$$

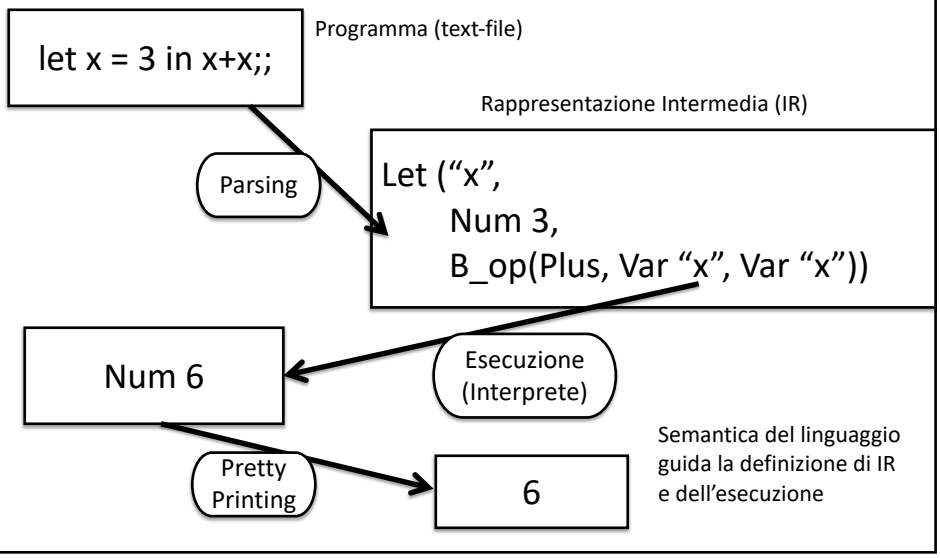
E’ utile?

Teorema
Sia E una espressione. E ⇒ v se e solo se eval(E) = v

Come si dimostra.
Per induzione strutturale sulla struttiura dell’albero di sintassi astratta.

Passiamo a linguaggi più articolati

La struttura



La struttura nel dettaglio

OCAML Type per descrivere la rappresentazione intermedia

```
type variable = string

type op = Plus | Minus | Times | …

type exp =
      | Int_e of int
      | Op_e of exp * op * exp
      | Var_e of variable
      | Let_e of variable * exp * exp

type value = exp
```

La struttura nel dettaglio

```
type variable = string

type op = Plus | Minus | Times | …

type exp =
      | Int_e of int
      | Op_e of exp * op * exp
      | Var_e of variable
      | Let_e of variable * exp * exp

type value = exp
```

Rappresentazione di "3 + 17"

```
let e1 = Int_e 3
let e2 = Int_e 17
let e3 = Op_e (e1, Plus, e2)
```

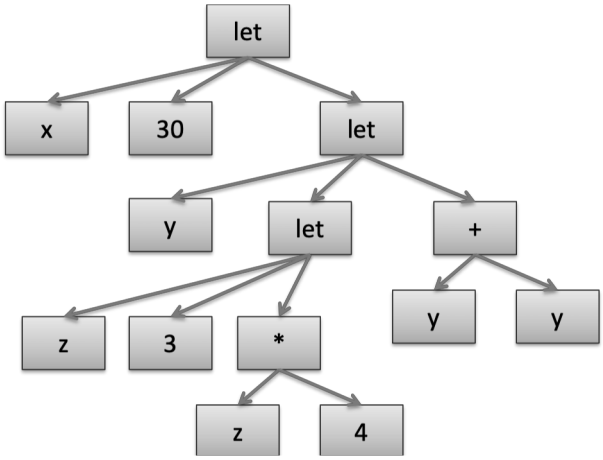
```
let x = 30 in
  let y =
   (let z = 3 in z*4) in
         y+y;;
```

Programma OCaml

Exp

```
Let_e("x", Int_e 30,
   Let_e("y",
      Let_e("z", Int_e 3,
               Op_e(Var_e "z", Times, Int_e 4)),
      Op_e(Var_e "y", Plus, Var_e "y")
```

AST



Variabili: dichiarazione e uso

```
type variable = string

type exp =
    | Int_e of int
    | Op_e of exp * op * exp
    | Var_e of variable
    | Let_e of variable * exp * exp
```

Run time: operazione di supporto

```
let is_value (e : exp) : bool =
    match e with
        | Int_e _ -> true
        | ( Op_e _
        | Let_e _
        | Var_e _ ) -> false;;

eval_op : value -> op -> value -> value

substitute : value -> variable -> exp -> exp
```

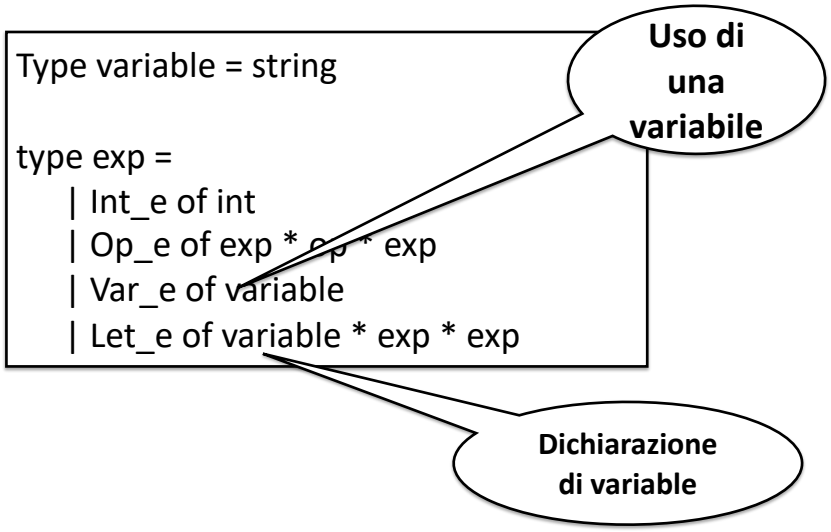
Variabili: dichiarazione e uso

```
Type variable = string

type exp =
    | Int_e of int
    | Op_e of exp * op * exp
    | Var_e of variable
    | Let_e of variable * exp * exp
```

Uso di una variabile

Variabili: dichiarazione e uso



L'interprete

RTS

```
is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e : exp) : exp =
    match e with
    | Int_e i ->
    | Op_e(e1,op,e2) ->
    | Let_e(x,e1,e2) ->
```

L'interprete

```
        is_value : exp -> bool
RTS     eval_op : value -> op -> value -> value
        substitute : value -> variable -> exp -> exp

        let rec eval (e : exp) : exp =
            match e with
            | Int_e i -> Int_e i
            | Op_e(e1,op,e2) ->
            | Let_e(x,e1,e2) ->
```

L'interprete

```
        is_value : exp -> bool
RTS     eval_op : value -> op -> value -> value
        substitute : value -> variable -> exp -> exp

        let rec eval (e : exp) : exp =
            match e with
            | Int_e i -> Int_e i
            | Op_e(e1,op,e2) -> let v1 = eval e1 in
                                let v2 = eval e2 in
                                  eval_op v1 op v2
            | Let_e(x,e1,e2) ->
```

L'interprete

```
        is_value : exp -> bool
RTS     eval_op : value -> op -> value -> value
        substitute : value -> variable -> exp -> exp

        let rec eval (e : exp) : exp =
            match e with
            | Int_e i -> Int_e i
            | Op_e(e1,op,e2) -> let v1 = eval e1 in
                                  let v2 = eval e2 in
                                    eval_op v1 op v2
            | Let_e(x,e1,e2) -> let v1 = eval e1 in
                                let e2' = substitute v1 x e2 in
                                  eval e2'
```

L'interprete

```
        is_value : exp -> bool
RTS     eval_op : value -> op -> value -> value
        substitute : value -> variable -> exp -> exp

        let rec eval (e : exp) : exp =
            match e with
            | Int_e i -> Int_e i
            | Op_e(e1,op,e2) -> eval_op eval e1 op eval e2
            | Let_e(x,e1,e2) -> let v1 = eval e1 in
                                let e2' = substitute v1 x e2 in
                                  eval e2'
```

L'interprete

RTS

```
is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e : exp) : exp =
    match e with
    | Int_e i -> Int_e i
    | Op_e(e1,op,e2) -> eval_op eval e1 op eval e2
    | Let_e(x,e1,e2) -> let v1 = eval e1 in
                        let e2' = substitute v1 x e2 in
                         eval e2'
```

Come avviene la valutazione?

Usare **let** per definirne l'ordine

L'interprete

RTS

```
is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e : exp) : exp =
    match e with
    | Int_e i -> Int_e i
    | Op_e(e1,op,e2) -> eval_op eval e1 op eval e2
    | Let_e(x,e1,e2) -> let v1 = eval e1 in
                        let e2' = substitute v1 x e2 in
                         eval e2'
    | Var_e x -> ???
```

Non dovremmo incontrare una variabile – l'avremmo già dovuta sostituire con un valore!!

Questo è un **errore di tipo**

L'interprete

RTS

```
is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e : exp) : exp option =
    match e with
    | Int_e i -> Some(Int_e i)
    | Op_e(e1,op,e2) -> eval_op eval e1 op eval e2
    | Let_e(x,e1,e2) -> let v1 = eval e1 in
                        let e2' = substitute v1 x e2 in
                         eval e2'
    | Var_e x -> None
```

Questo complica l'interprete: matching sui risultati delle chiamate ricorsive di eval

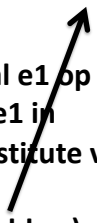
L'interprete

RTS

```
is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e : exp) : exp =
    match e with
    | Int_e i -> Int_e i
    | Op_e(e1,op,e2) -> eval_op eval e1 op eval e2
    | Let_e(x,e1,e2) -> let v1 = eval e1 in
                        let e2' = substitute v1 x e2 in
                         eval e2'
    | Var_e x -> raise (UnboundVariable x)
```

Tali eccezioni fanno parte del RTS



RTS: eval_op

```
let eval_op (v1:exp) (op:operand) (v2:exp) : exp =
    match v1, op, v2 with
        | Int_e i, Plus, Int_e j -> Int_e (i + j)
        | Int_e i, Minus, Int_e j -> Int_e (i - j)
        | Int_e i, Times, Int_e j -> Int_e (i * j)
        | _ , (Plus | Minus | Times), _ ->
          if is_value v1 && is_value v2
                        then raise TypeError
                        else raise NotValue
```

RTS: substitution

```
let substitute (v:value) (x:variable) (e:exp) : exp =
    let rec subst (e:exp) : exp =
        match e with
          | Int_e _ ->
          | Op_e(e1,op,e2) ->
          | Var_e y -> ... use x ...
          | Let_e (y,e1,e2) -> ... use x ...


    in subst e
```

RTS: substitution

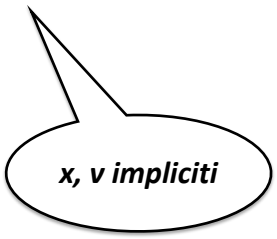
```
let substitute (v:value) (x:variable) (e:exp) : exp =
    let rec subst (e:exp) : exp =
        match e with
          | Int_e _ -> e
          | Op_e(e1,op,e2) ->
          | Var_e y -> ... use x ...
          | Let_e (y,e1,e2) -> ... use x ...

    in subst e
```

RTS: substitution

```
let substitute (v:value) (x:variable) (e:exp) : exp =
    let rec subst (e:exp) : exp =
        match e with
          | Int_e _ -> e
          | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)
          | Var_e y -> ... use x ...
          | Let_e (y,e1,e2) -> ... use x ...

    in subst e
```



RTS: substitution

```
let substitute (v:value) (x:variable) (e:exp) : exp =
    let rec subst (e:exp) : exp =
        match e with
          | Int_e _ -> e
          | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)
          | Var_e y -> if x = y then v else e
          | Let_e (y,e1,e2) -> ... use x ...

    in subst e
```

RTS: substitution

```
let substitute (v:value) (x:variable) (e:exp) : exp =
    let rec subst (e:exp) : exp =
        match e with
          | Int_e _ -> e
          | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)

          | Var_e y -> if x = y then v else e
          | Let_e (y,e1,e2) -> Let_e (y,subst e1,subst e2)

    in subst e
```

RTS: substitution

```
let substitute (v:value) (x:variable) (e:exp) : exp =
    let rec subst (e:exp) : exp =
        match e with
          | Int_e _ -> e
          | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)
          | Var_e y -> if x = y then v else e
          | Let_e (y,e1,e2) -> Let_e (y,subst e1,subst e2)

    in subst e
```

ERRORE
Se y=x

RTS: substitution

```
let substitute (v:value) (x:variable) (e:exp) : exp =
    let rec subst (e:exp) : exp =
        match e with
          | Int_e _ -> e
          | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)
          | Var_e y -> if x = y then v else e
          | Let_e (y,e1,e2) -> Let_e (y,subst e1,
                                 if x = y then e2 else subst e2)

    in subst e
```

If x = y …
shadow scope!!

Funzioni

Sintassi

```
type exp = Int_e of int | Op_e of exp * op * exp
    | Var_e of variable | Let_e of variable * exp * exp
    | Fun_e of variable * exp | FunCall_e of exp * exp
```

Sintassi

```
type exp = Int_e of int | Op_e of exp * op * exp
    | Var_e of variable | Let_e of variable * exp * exp
    | Fun_e of variable * exp | FunCall_e of exp * exp
```

La sintassi OCaml **fun x e** viene rappresentata come **Fun_e(x, e)**

La chiamata **fact 3** viene rappresentata come
FunCall_e (Var_e "fact", Int_e 3)

Estendiamo il RTS

```
let is_value (e : exp) : bool =
    match e with
        | Int_e _ -> true
        | Fun_e (_,_) -> true
        | ( Op_e (_,_,_)
            | Let_e (_,_,_)
            | Var_e _
            | FunCall_e (_,_) ) -> false
```

Le funzioni sono valori!!

Interprete ++

```
is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e : exp) : exp =
    match e with
    | :
    | Var_e x -> raise (UnboundVariable x)
    | Fun_e (x, e) -> Fun_e (x, e)
    | FunCall_e (e1, e2) ->
        match eval e1, eval e2 with
        | Fun_e (x, e), v2 -> eval (substitute v2 x e)
        | _ -> raise (TypeError)
```

Funzioni ricorsive

```
type exp = Int_e of int | Op_e of exp * op * exp
    | Var_e of variable | Let_e of variable * exp * exp
    | Fun_e of variable * exp | FunCall_e of exp * exp
    | Rec_e of variable * variable * exp
```

let g = rec f x -> f (x + 1) in g 3

```
Let_e ("g",
        Rec_e ("f", "x",
                FunCall_e (Var_e "f", Op_e (Var_e "x", Plus, Int_e 1))
               ),
        FunCall (Var_e "g", Int_e 3)
         )
```

Le funzioni ricorsive sono valori

```
let is_value (e : exp) : bool =
    match e with
        | Int_e _ -> true
        | Fun_e (_,_) -> true
        | Rec_e of (_,_,_) -> true
        | ( Op_e (_,_,_)
            | Let_e (_,_,_)
            | Var_e _
            | FunCall_e (_,_) ) -> false
```

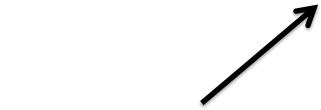
La nozione di sostituzione

- Sostituire il valore v al posto della variabile x nella espressione e: e [v / x]
- (x + y) [7/y] diventa (x + 7)
- (let x = 30 in let y = 40 in x + y) [7/y] diventa
  (let x = 30 in let y = 40 in x + y)
- (let y = y in let y = y in y + y) [7/y] diventa
  (let y = 7 in let y = y in y + y)

Come valutare le funzioni ricorsive

IDEA

(rec f x = body) arg --> body [arg/x] [rec f x = body/f]



Passaggio parametri

JIT compilation del body

```
let g = rec f x ->
    if x <= 0 then x
              else x + f (x - 1)
in g 3
```

La dichiarazione

```
g 3 [rec f x ->
        if x <= 0 then x
             else x + f (x-1) / g]
```

La sostituzione

```
(rec f x ->
    if x <= 0 then x else x + f (x - 1))
3
```

Risultato finale

```
(if x <= 0 then x else x + f (x - 1))
    [ 3 / x ]
    [ rec f x ->
        if x <= 0 then x
        else x + f (x - 1) / f ]
```

```
(if 3 <= 0 then 3 else 3 +
    (rec f x ->
        if x <= 0 then x
        else x + f (x - 1)) (3 - 1))
```

Interprete

```
is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e : exp) : exp =
    match e with
    | :
    | FunCall_e (e1, e2) ->
        match eval e1 with
          | Fun_e (x, e) -> let v = eval e2 in substitute e x v
          | (Rec_e (f, x, e)) as g -> let v = eval e2 in
                                      substitute (substitute e x v) f g
```

Cosa abbiamo imparato?

- OCaml può essere usato come linguaggio per la simulazione della semantica operazionale di un linguaggio (incluso se stesso!)
- Vantaggio: simulazione dell'implementazione
- Svantaggio: complicato per le operazioni da effettuare con i tipi di Ocaml
  - Op_e(e1, Plus, e2) rispetto a "e1 + e2"