PROGRAMMAZIONE 2
17.

Realizzare un interprete in OCaml

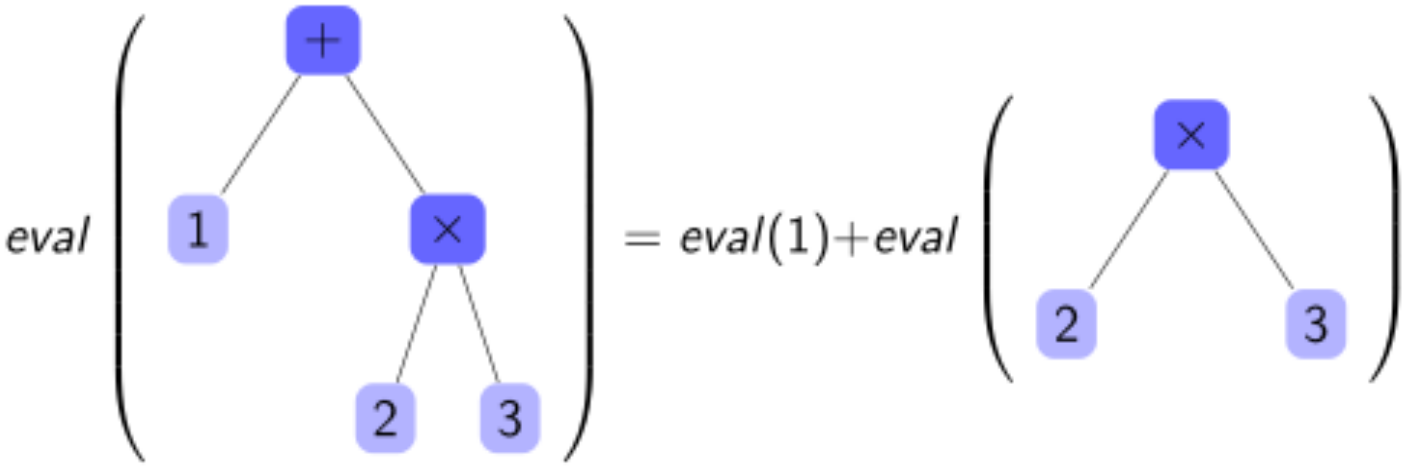
Valori e valutazione

- 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)` *?*
- *se* `e = n` *allora* `eval(e) = n` *(la costante* `n`*).*
- *se* `e = e1 + e2` *e* `eval(e1) =v1` *&* `eval(e2) =v2` *allora* `eval(e1 + e2) = v1 + v2`,
- *se* `e = e1 * e2` *e* `eval(e1) =v1` *&* `eval(e2) =v2` *allora* `eval(e1 * e2) = v1 * v2`

Graficamente



Graficamente (2)

$$eval(1)+eval\left(\begin{array}{ccc} & \times & \\ 2 & & 3 \end{array}\right) = eval(1)+(eval(2)\times eval(3))$$

$$eval(1) + (eval(2) \times eval(3)) = 1 + (2 \times 3) = 1 + 6 = 7$$

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.

$\mathbf{exp} \Rightarrow v$

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.

$$\mathbf{exp} \Rightarrow v$$

Valutazione di valori

$$\mathbf{v} \Rightarrow v$$

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.

$$\mathbf{exp} \Rightarrow v$$

Valutazione di operatori

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

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

Regole di valutazione

$$\mathbf{v} \Rightarrow v$$

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

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

OCAML: sintassi di espressioni

```
• 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)))
```

Interprete: espressioni

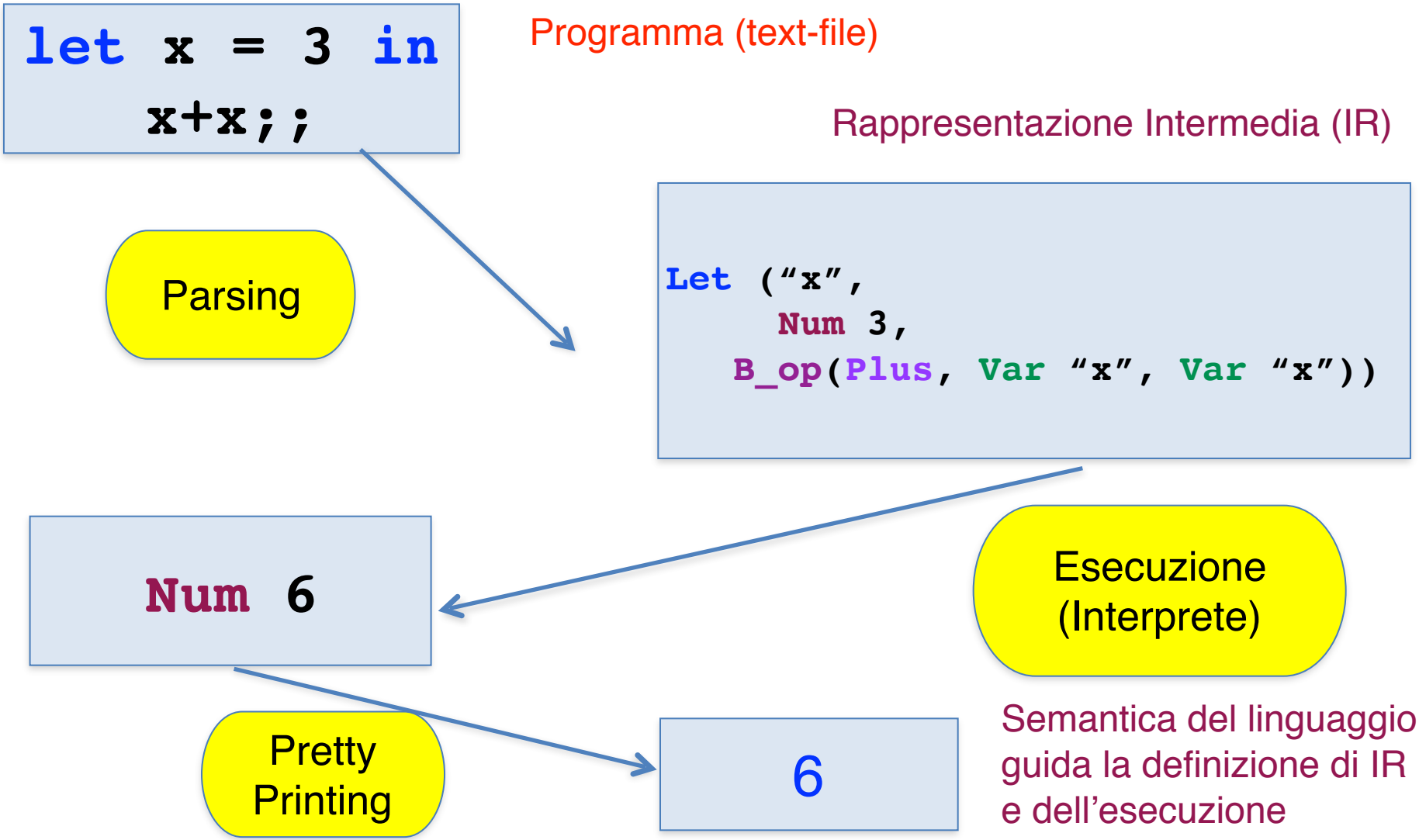
```
let rec eval (e : exp) : exp =
   match e with
    |Int_e i -> Int_e i
    |Op_e(e1,op,e2) -> let v1 = eval e1 in
                       let v2 = eval e2 in
                         eval_op v1 op v2
```

```
# 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
#
```

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;;
  ```

La struttura



La struttura nel dettaglio

OCAML Type per descrivere la rappresentazione intermedia

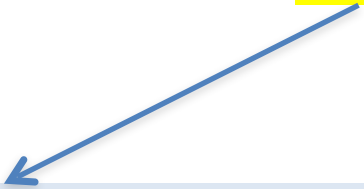
```
type variable = string
type op = Plus | Minus | Times | …
type exp =
    | Int_e of int
    | Op_e of exp * op * exp
    | Var_e of variable
    | Let_e of variable * exp *    exp

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
```

Rappresenta
“3 + 17”



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

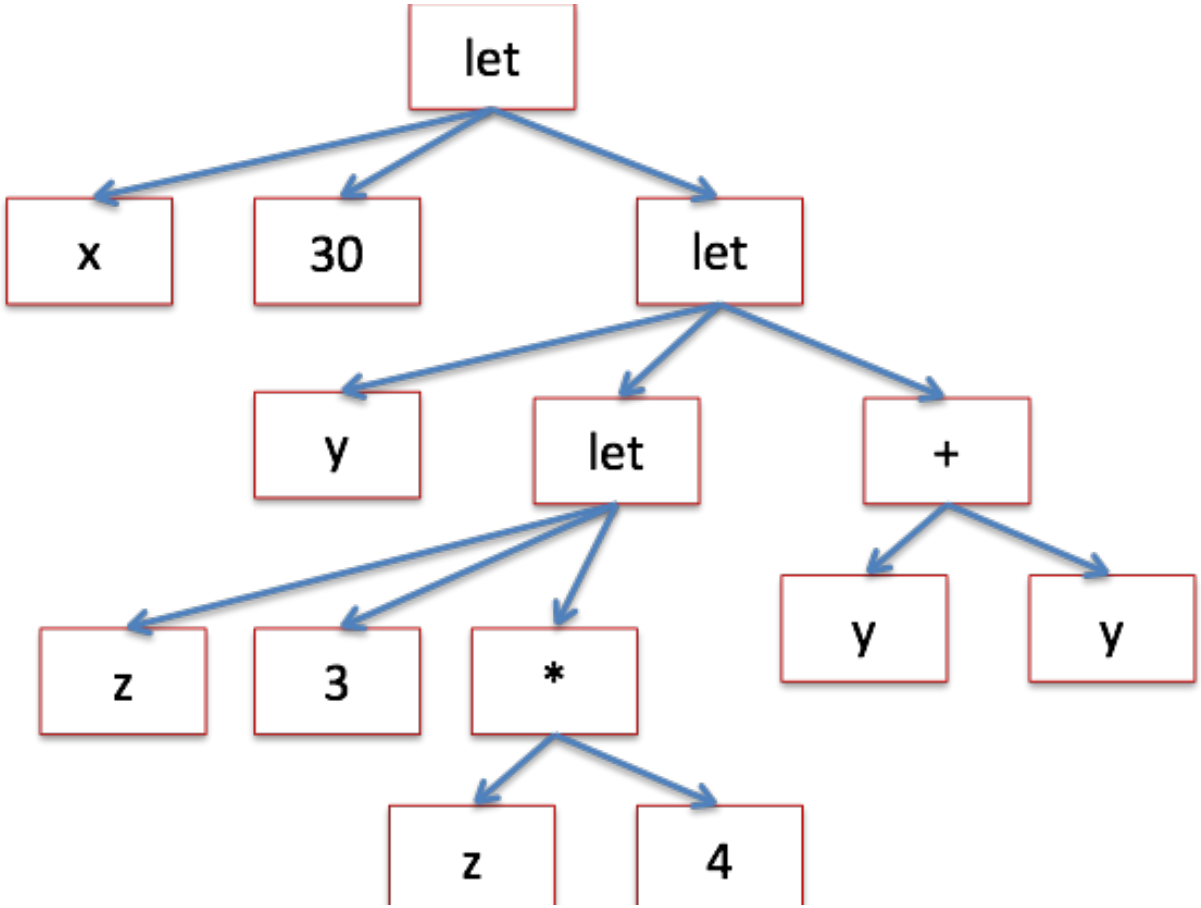
```
let x = 30 in
  let y =
   (let z = 3 in z*4) in
            y+y;;
```

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")
```

Albero della sintassi astratta: AST



Variabili: dichiarazione e uso

```
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
```

Variabili: dichiarazione e uso

```
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
```

Variabili: dichiarazione e uso

```
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
```

Uso di una variabile

Dichiarazione di variable

Espressioni: `let`

- Sia `e` l'espressione `let x = e1 in e2`
- Regola di valutazione
  - `eval (e1) = v1`
  - sostituiamo il valore `v1` per tutte le occorrenze di `x` in `e2` ottenendo l'espressione `e2'`
  - `subst(e2 , x, v1) = e2'`
  - `eval(e2') = v`
  - `eval(e) = v`

Regola di valutazione: `let`

$$\frac{eval(e1) = v1 \quad subst(e2, x, v1) = e2' \quad eval(e2') = v}{eval(let\ x = e1\ in\ e2) = v}$$

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
```

L’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
    | Int_e i ->
    | Op_e(e1,op,e2) ->
    | Let_e(x,e1,e2) ->
```

L’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
    | Int_e i -> Int_e i
    | Op_e(e1,op,e2) ->
    | Let_e(x,e1,e2) ->
```

L'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
    | 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

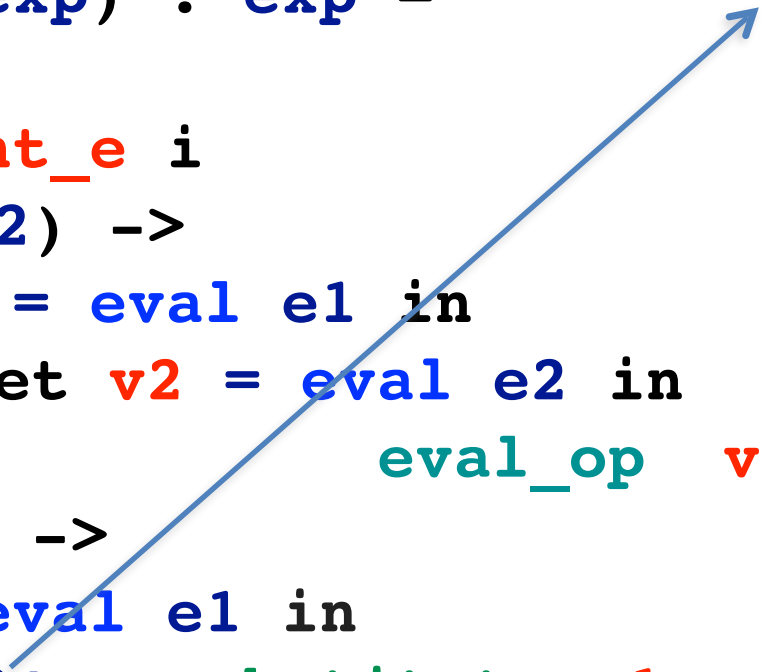
```
is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> exp
```

```
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

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

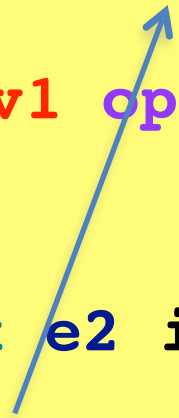
```
let rec eval (e : exp) : exp =
   match e with
     | Int_e i -> Int_e i
      | Op_e(e1,op,e2) ->
              let v1 = eval e1 in
                   let v2 = eval e2 in
                                eval_op  v1 op v2
    |Let_e(x,e1,e2) ->
          let v1 = eval e1 in
               let e2'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



L'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(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
```

Esempi: 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)`

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

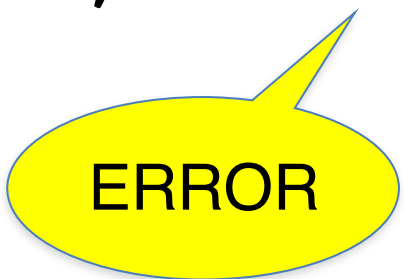
```
let substitute (v:value) (x:variable)
                          (e:exp) : exp =
  let rec subst (e:exp) : exp =
    match e with
      | Int_e _ -> e
      | Op_e(e1,op,e2)  ->
            Op_e(subst e1,op,subst e2)
      | Var_e y -> if x = y then v else e
      | Let_e (y,e1,e2) -> ... use x ...


  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,
            if x = y then e2 else subst e2)
 in subst e
```

Sintassi:funzioni

```
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
    | Fun_e of variable *  exp
    | FunCall_e of exp * exp

type value = exp
```

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

La chiamata **`inc 3`** viene rappresentata come
`FunCall_e (Var_e “inc”, 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 _ ) -> false
       | 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
    | Int_e i -> Int_e i

    |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

```
ype 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
     | Fun_e of variable *  exp
     | FunCall_e of exp * exp
     | Rec_e of variable * variable * exp


type value = exp
```

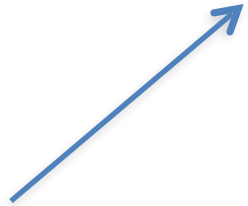
Esempio: funzioni ricorsive

```
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_e(Var_e "g", Int_e 3))
```

Come valutare le funzioni ricorsive?

```
( 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

La sostituzione

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

Risultato finale

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

```
(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 v2 = eval e2 in
            eval (substitute v2 x e)
   |Rec_e (f, x, e) as g ->
            let v2 = eval e2 in
  eval (substitute (substitute v2 x e) f g)
| _ -> raise (BadArgumentError)
```

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`