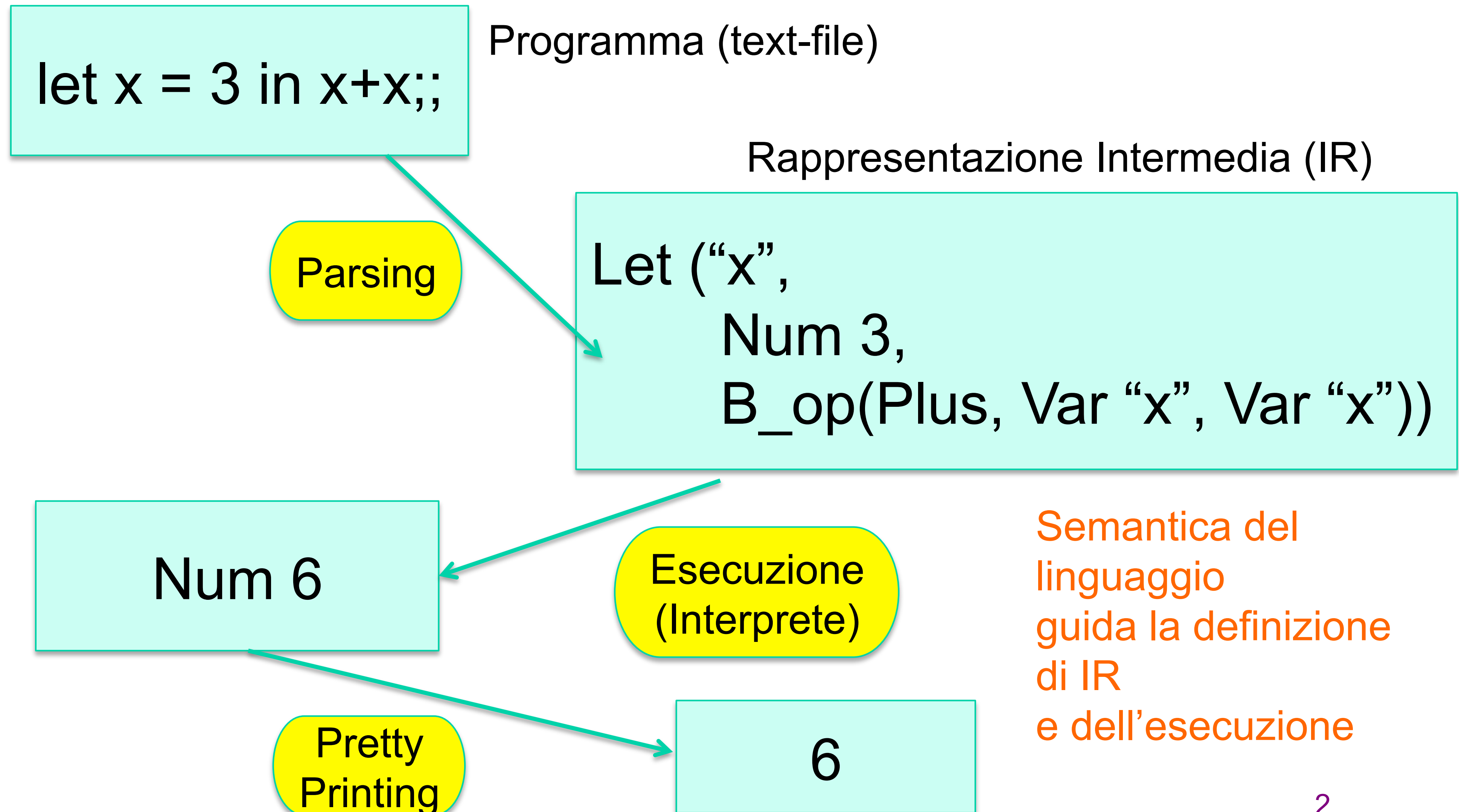
PROGRAMMAZIONE 2
16. Realizzare un interprete in OCaML

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
```

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
```

Rappresentazione di "3 + 17"

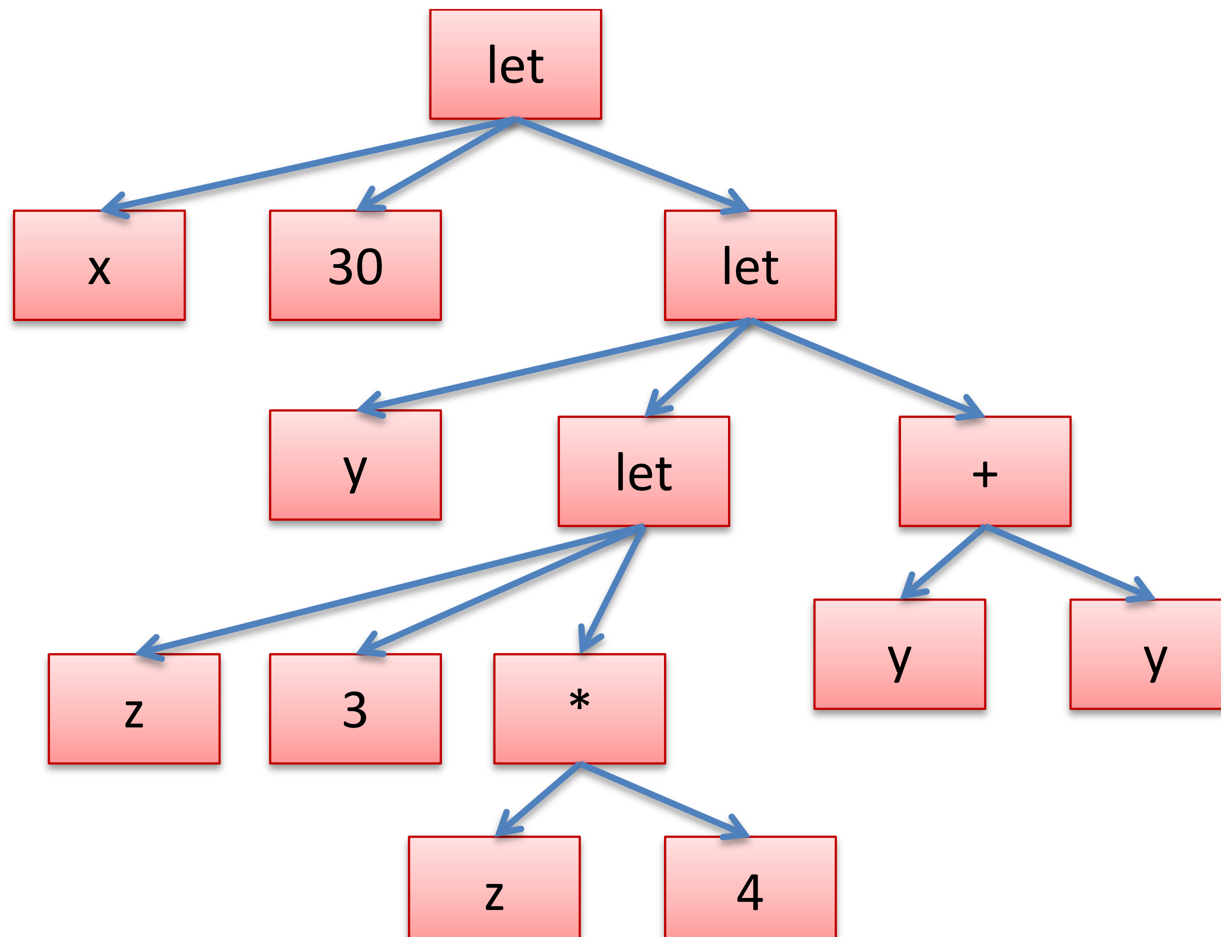
```
let e1 = Int_e 3
let e2 = Int_e 17
let e3 = Op_e (e1, Plus, e2)
```

```
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")
```


let
x
30
let
y
let
+
z
3
*
y
y
z
4

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

Dichiarazione di variable

Runtime: operazione di supporto

eval_op : exp **->** **op** **->** exp **->** exp

substitute : exp **->** variable **->** exp **->** exp

L'interprete

RTS

eval_op : exp**->** **op** **->** exp**->** exp

substitute : exp**->** variable **->** exp**->** exp

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

L'interprete

eval_op : exp**->** **op** **->** exp**->** exp
substitute : exp**->** variable **->** exp**->** exp

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

L'interprete

eval_op : exp-> **op** -> exp-> exp
substitute : exp-> variable -> exp-> exp

```
let rec eval (e : exp) : exp =
      match e with
       Int_e _ -> e
      | 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

eval_op : exp-> **op** -> exp-> exp
substitute : exp-> variable -> exp-> exp

```
let rec eval (e : exp) : exp =
  match e with
    Int_e _ -> e
   | 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

eval_op : exp-> **op ->** exp-> exp
substitute : exp-> variable **->** exp-> exp

```
let rec eval (e : exp) : exp =
    match e with
       Int_e _ -> e
      | 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

eval_op : exp-> **op ->** exp-> exp
substitute : exp-> variable **->** exp-> exp

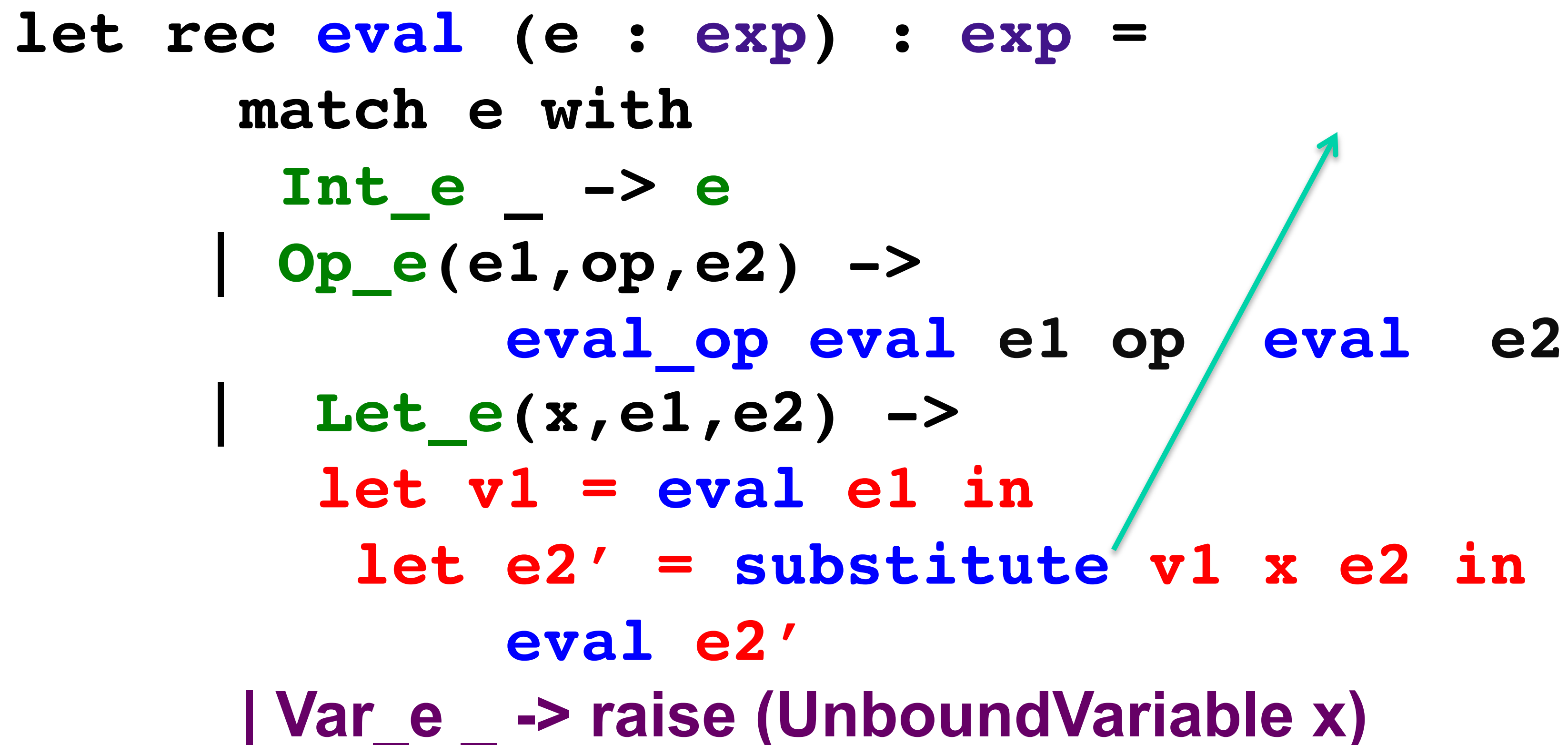
```
let rec eval (e : exp) : exp =
   match e with
       Int_e _ -> e
     | Op_e(e1,op,e2) ->
           eval_op eval e1 op  eval  e2
     |  Let_e(x,e1,e2) ->
        let v1 = eval e1 in
         let e2' = substitute v1 x e2 in
           eval e2'

     | Var_e _ -> ???
```

Non dovremmo incontrare una variabile – avremmo già dovuta sostituirla con un valore!!

Questo è un **errore di tipo**

L’interprete



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)
       | _, _, _ -> raise (BadOp (v1,op,v2))
```

RTS: substitute

```
let substitute (v:exp) (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:exp) (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:exp) (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
```

Impliciti x,v

RTS: substitution

```
let substitute (v:exp) (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:exp) (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 e_2)
     in subst e
```

Errore se x=y!!

RTS: substitution

```
let substitute (v:exp) (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
```

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 chiamata **fact 3** viene rappresentata come
FunCall_e (Var_e “fact”, Int_e 3)

L’interprete+

eval_op : exp**->** **op** **->** exp**->** exp
substitute : exp**->** variable **->** exp**->** exp

```
let rec eval (e : exp) : exp =
      match e with
      …..
     |Fun_e _ -> e
     |FunCall_e (e1,e2) ->
         match eval e1, eval e2 with
              Fun_e (x,e3), v2 ->
              eval (substitute v2 x e3)
             | _ -> raise (TypeError)
```

Ricorsione

```
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
    | Letrec_e of variable * exp * exp
```

L'interprete++

eval_op : exp**->** **op** **->** exp**->** exp
substitute : exp**->** variable **->** exp**->** exp

```
let rec eval (e : exp) : exp =
     match e with
…….
     |Letrec_e(x,e1,e2) ->
        let e1_unwind =
 substitute (Letrec_e (x,e1,Var_e x)) x e1
in  eval (Let_e (x,e1_unwind,e2))
```

Concludendo

- 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"