1

SEMANTICA OPERAZIONALE ESEGUIBILE

OCAML

- Nucleo funzionale puro
  - funzioni (ricorsive)
  - tipi e pattern matching
  - primitive utili: liste
- Componente imperativo
  - variabili e assegnamento
  - primitive utili: arrays
- Moduli e oggetti
- Fortemente tipato:
  - Type checking e type inference

2

OCAML

- Perche' ci interessa?
  - Linguaggio di programmazione completo e moderno
    - ✓contiene tutte quello che normalmente serve e anhe altre cose utili
  - Linguaggio intermedio per programmare l'interprete del linguaggio didattico
    - ✓trascriviamo in OCAML pezzi della semantica operazionale

3

Operare con le funzioni

```
# function x -> x + 1;;
- : int -> int = <fun>
# (function x -> x + 1) 3;;
- : int = 4
# (function x -> x + 1) true;;
This expression has type bool but is here used with type int
# function x -> x;;
- : 'a -> 'a = <fun>
# function x -> function y -> x y;;
- : ('a -> 'b) -> 'a -> 'b = <fun>
# (function x -> x) 2;;
- : int = 2
# (function x -> x) (function x -> x + 1);;
- : int -> int = <fun>
# function (x, y) -> x + y;;
- : int * int -> int = <fun>
# (function (x, y) -> x + y) (2, 33);;
- : int = 35
```

4

Dichiarazioni: Let binding

```
# let x = 3;;
val x : int = 3
# x;;
- : int = 3
# let y = 5 in x + y;;
- : int = 8
# y;;
Unbound value y
# let f = function x -> x + 1;;
val f : int -> int = <fun>
# f 3;;
- : int = 4
# let f x = x + 1;;
val f : int -> int = <fun>
# f 3;;
- : int = 4
# let fact x = if x = 0 then 1 else x * fact(x - 1) ;;
Unbound value fact
```

5

Let binding

- let x = 1
  let y = x + 1
  let x = 1000
  let z = x + 2
  let test () : bool = z = 1002
  ;; run_test “x shadowed” test

6

Dichiarazioni ricorsive: Let rec binding


```
# let rec fact x = if x = 0 then 1 else x * fact(x - 1) ;;
val fact : int -> int = <fun>
# fact (x + 1);;
- : int = 24
```

7

Un tipo primitivo utile: le liste


```
# let l1 = [1; 2; 1];;
val l1 : int list = [1; 2; 1]
# let l2 = 3 :: l1;;
val l2 : int list = [3; 1; 2; 1]
# let l3 = l1 @ l2;;
val l3 : int list = [1; 2; 1; 3; 1; 2; 1]
# List.hd l3;;
- : int = 1
# List.tl l3;;
- : int list = [2; 1; 3; 1; 2; 1]
# List.length l3;;
- : int = 7
```

8

Map

- Funzione utile per operare con liste
- let rec map (f: ‘a -> ‘b) (l: ‘a list) : ‘b list =
  begin match l with
  | [ ] -> [ ]
  | h :: t -> (f h) :: (map f t)
  end
- Posso scrivere funzionali simili per operare con altre strutture dati
  map_tree : (‘a -> ‘b) -> tree ‘a -> tree ‘b

9

TIPI E SINTASSI ASTRATTA

10

Tipi 1


```
# type ide = string;;
type ide = string
# type expr =  | Den of ide  | Val of ide  | Fun of ide * expr
  | Plus of expr * expr  | Apply of expr * expr;;
type expr =
  | Den of ide
  | Val of ide
  | Fun of ide * expr
  | Plus of expr * expr
  | Apply of expr * expr
```

E ::= I | val(I) | lambda(I, E_1) | plus(E_1, E_2) | apply(E_1, E_2)

```
# Apply(Fun("x",Plus(Den "x", Den "x")), Val "y");;
- : expr = Apply (Fun ("x", Plus (Den "x", Den "x")), Val "y")
```


Tipi 2


```
# type eval = Int of int | Bool of bool | Efun of expr
    | Unbound;;
type eval = | Int of int | Bool of bool | Efun of expr |
    Unbound
# type env = ide -> eval;;
type env = ide -> eval
# let bind (rho, i, d) =
        function x -> if x = i then d else rho x;;
val bind : (ide -> eval) * ide * eval -> (ide -> eval) = <fun>
```

- env = IDE → eval
- eval = [int + bool + fun]

Tipi 3

```
# type com = Assign of ide * expr | Ifthenelse of expr *
  com list * com list | While of expr * com list;;
type com =
  | Assign of ide * expr
  | Ifthenelse of expr * com list * com list
  | While of expr * com list
```

C ::= ifthenelse(E, C_1, C_2) | while(E, C_1) | assign(I, E) | cseq(C_1, C_2)

```
# While(Den "x", [Assign("y", Plus(Val "y", Val "x"))]);;
- : com = While (Den "x", [Assign ("y", Plus (Val "y", Val
    "x"))])
```


Tipi e pattern matching

```
type expr =
  | Den of ide
  | Fun of ide * expr
  | Plus of expr * expr
  | Apply of expr * expr
type eval = | Int of int | Bool of bool | Efun of expr | Unbound
type env = ide -> eval
```

$\mathcal{E}$(I, ρ) = ρ(I)
$\mathcal{E}$(plus(E_1, E_2), ρ)= $\mathcal{E}$(E_1, ρ) + $\mathcal{E}$(E_2, ρ)
$\mathcal{E}$(lambda(I, E_1), ρ)= lambda(I, E_1)
$\mathcal{E}$(apply(E_1, E_2), ρ) = applyfun ($\mathcal{E}$(E_1, ρ), $\mathcal{E}$(E_2, ρ), ρ)
applyfun(lambda(I, E_1), d, ρ) = $\mathcal{E}$(E_1) [ρ / I ← d]

```
# let rec sem (e, rho) = match e with
    | Den i -> rho i
    | Plus(e1, e2) -> plus(sem (e1, rho), sem (e2, rho))
    | Fun(i, e) -> Efun(Fun(i, e))
    | Apply(e1, e2) -> match sem(e1, rho) with
         | Efun(Fun(i, e))  -> sem(e, bind(rho, i, sem(e2, rho))
         | _ -> failwith("wrong application");;
val sem : expr * env -> eval = <fun>
```

Variabili e frammento imperativo

```
# let x = ref(3);;
val x : int ref = {contents=3}
# !x;;
- : int = 3
# x := 25;;
- : unit = ()
# !x;;
- : int = 25
# x := !x + 2; !x;;
- : int = 27
```


Un tipo primitivo mutabile: l'array

```
# let a = [| 1; 2; 3 |];;
val a : int array = [|1; 2; 3|]
# let b = Array.make 12 1;;
val b : int array = [|1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1|]
# Array.length b;;
- : int = 12
# Array.get a 0;;
- : int = 1
# Array.get b 12;;
Uncaught exception: Invalid_argument("Array.get")
# Array.set b 3 131;;
- : unit = ()
# b;;
- : int array = [|1; 1; 1; 131; 1; 1; 1; 1; 1; 1; 1; 1|]
```

Moduli: interfaccie

```
# module type PILA =
   sig
      type 'a stack               (* abstract *)
     val emptystack : 'a stack
     val push : 'a stack -> 'a -> 'a stack
     val pop : 'a stack -> 'a stack
     val top : 'a stack -> 'a
   end;;
module type PILA =
  sig
    type 'a stack
    val emptystack : 'a stack
    val push : 'a stack -> 'a -> 'a stack
    val pop : 'a stack -> 'a stack
    val top : 'a stack -> 'a
  end
```

17

Moduli: implementazione

```
# module SpecPila: PILA =
   struct
      type 'a stack = Empty | Push of 'a stack * 'a
     let emptystack = Empty
     let push p a = Push(p,a)
     let pop p = match p with
        | Push(p1, _) -> p1
     let top p = match p with
        | Push(_, a) -> a
   end;;
module SpecPila : PILA
```

18

Il linguaggio didattico

- le cose semanticamente importanti di OCAML, meno
  - tipi (e pattern matching)
  - moduli
  - eccezioni

19