Implementing a Functional Language

A Basic Functional Language

- Goal: illustrate and understand the features of the run-time support

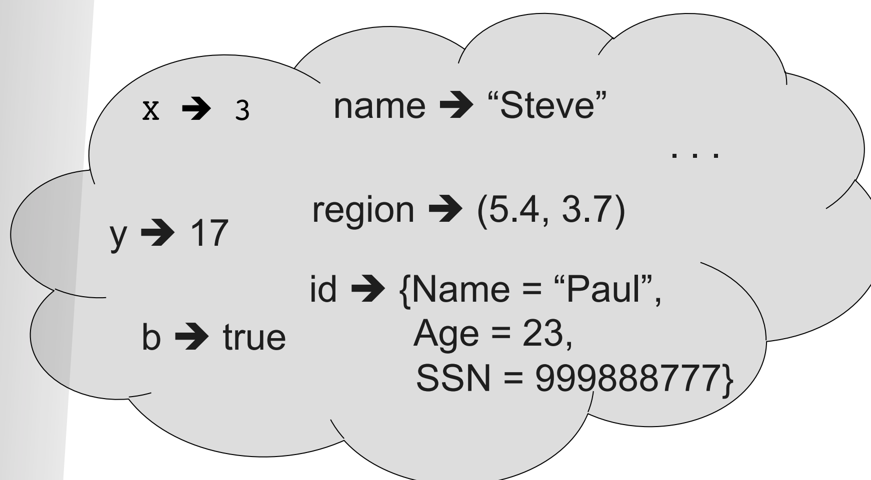
Environment

- *Environments* record what value is associated with a given identifier
- Central to the semantics and implementation of a programming language
- Notation

  env = {$name_1 \rightarrow value_1$, $name_2 \rightarrow value_2$, …}

  Using set-like notation, but describes a partial function
- Often stored as list, or stack
  - To find value start from left and take first match

3

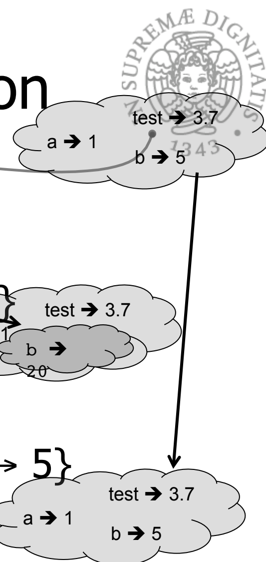


4

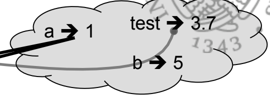
```
# 2 + 3;;     (* Expression *)
// doesn't affect the environment
# let test = 3 < 2;;       (* Declaration *)
val test : bool = false
//  env1 = {test → false}
# let a = 1 let b = a + 4;; (* Seq of dec *)
//  env2 = {b → 5, a → 1, test → false}
```


Local Variable Creation

```
// ρ3 = {test → 3.7, a → 1, b → 5}
# let b = 5 * 4
// ρ4 = {b → 20, test → 3.7, a → 1}
   in 2 * b;;
- : int = 40
// ρ5 = ρ3= {test → 3.7, a → 1, b → 5}
# b;;
- : int = 5
```

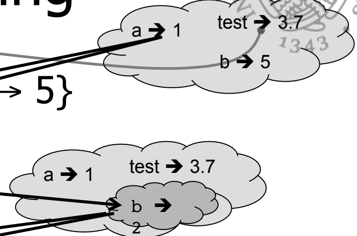


Local let binding



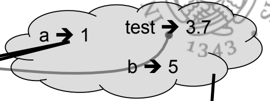
// ρ_5 = {test → 3.7, a → 1, b → 5}
let c =
    let b = a + a
// ρ_6 = {b → 2} + ρ_3
//   ={b → 2, test → 3.7, a → 1}
    in b * b;;
val c : int = 4
// ρ_7 = {c → 4, test → 3.7, a → 1, b → 5}
b;;
- : int = 5

Local let binding

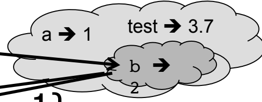


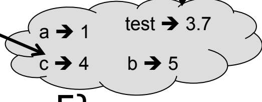
// ρ_5 = {test → 3.7, a → 1, b → 5}
let c =
    let b = a + a
// ρ_6 = {b → 2} + ρ_3
//   ={b → 2, test → 3.7, a → 1}
    in b * b;;
val c : int = 4
// ρ_7 = {c → 4, test → 3.7, a → 1, b → 5}
b;;
- : int = 5

Local let binding



// ρ_5 = {test → 3.7, a → 1, b → 5}
let c =
    let b = a + a
// ρ_6 = {b → 2} + ρ_3
//   ={b → 2, test → 3.7, a → 1}
    in b * b;;
val c : int = 4
// ρ_7 = {c → 4, test → 3.7, a → 1, b → 5}
b;;
- : int = 5





Environment

```
(* Environment *)


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;;
```

The Language

```
type ide = string
type exp =  Eint of int
    | Ebool of bool
    | Var of ide
    | Prod of exp * exp
    | Sum of exp * exp
    | Diff of exp * exp
    | Eq of exp * exp
    | Minus of exp
    | Iszero of exp
    | Or of exp * exp
    | And of exp * exp
    | Not of exp
    | Ifthenelse of exp * exp * exp
    | Let of ide * exp * exp
    | Fun of ide list * exp
    | Appl of exp * exp list
```

```
let rec eval((e: exp), (r: env)) =
      match e with
      | Eint(n) -> Int(n)
      | Ebool(b) -> Bool(b)
      | Var(i) -> lookup(r, i)
      | Iszero(a) -> iszero(eval(a, r))
      | Eq(a, b) -> equ(eval(a, r),eval(b, r))
      | Prod(a, b) ->  mult(eval(a, r), eval(b, r))
      | Sum(a, b) ->  plus(eval(a, r), eval(b, r))
      | Diff(a, b)  ->  diff(eval(a, r), eval(b, r))
      | Minus(a) ->  minus(eval(a, r))
      | And(a, b) ->  et(eval(a, r), eval(b, r))
      | Or(a, b) ->  vel(eval(a, r), eval(b, r))
      | Not(a) -> non(eval(a, r))
      | Ifthenelse(a, b, c) -> let g = eval(a, r) in
            if typecheck("bool", g) then
               (if g = Bool(true) then eval(b, r) else eval(c, r))
            else failwith ("nonboolean guard")
      |
```

Operational evalantics

$$\frac{env \triangleright e1 \Rightarrow v1 \qquad env[v1/x] \triangleright e2 \Rightarrow v2}{env \triangleright Let(x, e1, e2) \Rightarrow v2}$$

13

```
let rec eval ((e: exp), (r: env)) =
      match e with
        | Eint(n) -> Int(n)
        | Ebool(b) -> Bool(b)
        | Var(i) -> lookup(r, i)
        | Iszero(a) -> iszero(eval(a, r))
        | Eq(a, b) -> equ(eval(a, r),eval(b, r))
        | Prod(a, b) ->  mult(eval(a, r), eval(b, r))
        | Sum(a, b) ->  plus(eval(a, r), eval(b, r))
        | Diff(a, b)  ->  diff(eval(a, r), eval(b, r))
        | Minus(a) ->  minus(eval(a, r))
        | And(a, b) ->  et(eval(a, r), eval(b, r))
        | Or(a, b) ->  vel(eval(a, r), eval(b, r))
        | Not(a) -> non(eval(a, r))
        | Ifthenelse(a, b, c) -> let g = eval(a, r) in
              if typecheck("bool", g) then
                 (if g = Bool(true) then eval(b, r) else eval(c, r))
              else failwith ("nonboolean guard")
        | Let(i, e1, e2) ->
                 eval(e2, bind (r, i, eval(e1, r)))
```

14

Example

```
# eval(Let("x", Sum(Eint 1, Eint 0),
      Let("y", Ifthenelse(Eq(Den "x", Eint 0),
                          Diff(Den "x", Eint 1),
                          Sum(Den "x", Eint 1)),
          Let("z", Sum(Den "x", Den "y"), Den "z"))),
      (emptyenv));;

-: eval = Int 3
```

OCAML Syntax

```
# let x = 1+0 in
   let y = if x = 0 then x-1 else x+1 in
    let z = x + y in z;;

-: int = 3
```

15

Functions

- Functional abstraction and Application

16

Syntax

```
type exp =  ...
      | Fun of ide list * exp
      | Appl of exp * exp list
```

Abstractio

Application

17

Funzioni

- Formal Parameters

  Fun of ide list * exp

- Actual Parameters

  Appl of exp * exp list

18

Scoping

- Assume static scoping

```
type eval = | Int of int | Bool of bool |
   Unbound
             | Funval of efun
      and efun = exp * eval env
```

- Functional abstraction is a closure
  - Code of functions
  - Environment (when the functions has been declared)

  Non local references are solved by taking advantage of the closure

19

Operational semantics

$$env \triangleright Fun(x,e) \Rightarrow Funval(Fun(x,e),\ env)$$

$$\frac{env \triangleright e1 \Rightarrow v1 \qquad v1 = Funval(Fun(x,e),\ env1) \qquad env \triangleright e2 \Rightarrow v2 \qquad env1[v2/x] \triangleright e \Rightarrow v}{env \triangleright Apply(e1,e2) \Rightarrow v}$$

20

eval

```
let rec eval ((e: exp), (r: eval env)) =
   match e with
     | ...
     | Fun(x, a) -> Funval(e, r)
     | Apply(e1, e2) -> match eval(e1, r) with
         | Funval(Fun(x, a), r1) ->
             eval(a, bind(r1, x, eval(e2, r)))
         | _ -> failwith("no funct in apply")
```

21

$$\frac{env \triangleright e1 \Rightarrow v1 \qquad v1 = Funval(Fun(x,e), env1) \qquad env \triangleright e2 \Rightarrow v2 \qquad env1[v2/x] \triangleright e \Rightarrow v}{env \triangleright Apply(e1,e2) \Rightarrow v}$$

```
let rec eval ((e: exp), (r: eval env)) =
   match e with
     | ...
     | Fun(x, a) -> Funval(e, r)
     | Apply(e1, e2) -> match eval(e1, r) with
         | Funval(Fun(x, a), r1) ->
             eval(a, bind(r1, x, eval(e2, r)))
         | _ -> failwith("no funct in apply")
```

22

Dynamic scope

```
type eval = | Int of int | Bool of bool | Unbound
            | Funval of efun
and efun = expr
```

efun contains only the code of the function

The function will be evaluated in the environment of the call.

23

Operational semantics

$$env \triangleright Fun(x,e) \Rightarrow Funval(Fun(x,e))$$

$$\frac{env \triangleright e1 \Rightarrow v1 \qquad v1 = Funval(Fun(x,e)) \qquad env \triangleright e2 \Rightarrow v2 \qquad env[v2/x] \triangleright e \Rightarrow v}{env \triangleright Apply(e1,e2) \Rightarrow v}$$

24

Eval

```
let rec eval (e: exp) (r: eval env) =
   match e with
     | ...
     | Fun("x", a) -> Funval(e)
     | Apply(e1, e2) -> match eval(e1, r) with
            | Funval(Fun("x", a)) ->
              eval(a, bind(r, x, eval(e2, r)))
            | _ -> failwith("no funct in apply")
```

25

Scoping rule

```
type efun = expr * eval env
|Apply(e1, e2) -> match eval(e1, r) with
         | Funval(Fun("x", a), r1) ->
           eval(a, bind(r1, x, eval(e2, r)))
```

- **Static scoping**:

```
type efun = expr
| Apply(e1, e2) -> match eval(e1, r) with
         | Funval(Fun("x", a)) ->
           eval(a, bind(r, x, evall(e2, r)))
```

- **Dynamic scoping**:

26