Esami

1) $T \triangleright \text{BothFun}(x, e_1, e_2) \Rightarrow \text{BothClosure}(x, e_1, e_2, T)$

$$\frac{T \triangleright e_1 \Rightarrow \text{BothClosure}(arg, body_1, body_2, T_{fDecl}) \quad T \triangleright e_2 \Rightarrow aVal \quad T_{fDecl}[arg = aVal] \triangleright body_1 \Rightarrow v_1 \quad T_{fDecl}[arg = aVal] \triangleright body_2 \Rightarrow v_2}{T \triangleright \text{Apply}(e_1, e_2) \Rightarrow (v_1, v_2)}$$

```
type exp = ...
    | BothFun of ide * exp * exp

type evT = ...
    | BothClosure of ide * exp * exp * evT env
    | Pair of exp * exp

let rec eval e s =
    match e with ...
    | BothFun(arg, body1, body2) →
        BothClosure(arg, body1, body2, s)
    | Apply(e1, e2) →
        let fclosure = eval e1 s in
        (match fclosure with ...
        | BothClosure(arg, body1, body2, fDecEnv) →
            let aval = eval e2 s in
            let aenv = bind fDecEnv arg aVal in
            Pair (eval body1 aenv, eval body2 aenv))
```

2) type tname = ... | TCoda

```
type exp = ...
        | CodaLimitata of exp
        | Insert of exp * exp
        | Remove of exp * exp
        | Peek of exp

type evT = ...
        | Coda of evT list * int

let typecheck (x,y) =
        match (x,y) with
        |(TCoda, Coda(lis,n)) -> true
        |(_,_) -> false

let rec eval e s =
    match e with...
    | CodaLimitata e1 -> let size = eval e1 s in
                         if typecheck(TInt, size) then
                         (match size with
                         | Int(n) -> Coda([],n)
                         | _ -> failwith "Errore")
    |Insert (elem, e1) -> let coda = eval e1 s in
                          (match coda with
                          | Coda(lis, size) ->
                              if List.length lis < size then
                              let x = eval elem s in
                              Coda(lis@[x], size)
                              else failwith "Coda piena"
```

```
                                | _ -> failwith "Errore" )
| Remove(elem, e1) -> let coda = eval e1 s in
                      (match coda with
                      | Coda([], size) ->
                          failwith "Coda vuota"
                      | Coda(x::lis, size) ->
                          Coda(lis, size)
                      | _ -> failwith "Errore")
| Peek e1 -> let code = eval e1 s in
             (match coda with
             | Coda([], size) ->
                 failwith "Code vuota"
Alternativa
// Coda(x::lis, size) -> x   | Coda(lis, size) ->
                              List.hd lis
                          | _ -> failwith "Error")
```

3) Pixel $p ::= \langle r, g, b \rangle$ dove $r, g, b \in \{0, \ldots, 255\}$
Identificatori $I ::= \ldots$
Espressioni $e ::= I \mid p \mid$ lighten e | darken e | let I = e1 in e2

```
type pixel = int * int * int

type evT = Unbound | Epix of pixel

type ide = string

type exp = Ide of ide | Pix of pixel
         | Light of exp | Darek of exp
         | Let of ide * exp * exp
```

```
let check a = a>-1 && a<256

let inc a = match a with
    | 255 -> 255
    | n -> n+1

let dec a = match a with
    | 0 -> 0
    | n -> n-1

let increase v = match v with
    | Epix(a,b,c) -> Epix(inc a, inc b, inc c)

let decrease v = match v with
    | Epix(a,b,c) -> Epix(dec a, dec b, dec c)

let rec eval e s =
    match e with ...
    | Ide(i) -> s i
    | Pix(a,b,c) -> if check a && check b && check c then
                        Epix(a,b,c) else UnBound
    | Light e1 -> let v = eval e1 s in increase v
    | Dark e1 -> let v = eval e1 s in decrease v
    | Let(i, e1, e2) -> let v = eval e1 s in let s1 = bind
```

4) for-each (lista-interi ; funzione)

```
type exp = ...
        | ForEach of exp list * exp

let rec eval e s =
    match e with ...
    | ForEach e1 e2 -> (let lis = evalSeq e1 s in
                        let f = eval e2 s in
                        match (lis, f) with
                        | (lst, Closure (i, e, s1)) ->
                              Int (evdList lst (i, e, s1) s)
                        | (_, _) -> failwith "Errore")
and evalSeq lst s =
    match lst with
    | [] -> []
    | e :: lst' -> let x = eval e s in
                   (match (typecheck (TInt, x), x) with
                   | (true, Int v) -> v :: (evalSeq lst' s)
                   | (false, _) -> failwith "Errore")

and evalList lst funB s =
    match (lst, funB) with
    | ([], _) -> 0
    | (v :: lst', (arg, fBody, fDecEnv)) ->
          match (eval fbody (bind fDecEnv arg (Int v))) with
          | Int v -> v' + (evdList lst' funB s) ;;
```

5)

$$\frac{T \triangleright e \Rightarrow v \quad T' = T[f \mapsto \text{MyClosure}(x, v, e_1, T)] \quad T' \triangleright e_2 \Rightarrow v_1}{T \triangleright \text{function } f(x = e) = e_1 \text{ in } e_2 \Rightarrow v_1}$$

$$\frac{T(f) = \text{MyClosure}(x, v, e, T_{fdecl}) \quad T \triangleright e_{arg} \Rightarrow v_{arg} \quad T_{fdecl}[x \mapsto v_{arg}] \triangleright e \Rightarrow v_1}{T \triangleright f(e_{arg}) \Rightarrow v_1}$$

$$\frac{T(f) = \text{MyClosure}(x, v, e, T_{fdecl}) \quad T_{fDecl}[x \to v] \triangleright e \Rightarrow v_1}{T \triangleright f() \Rightarrow v_1}$$

```
type exp = ...
    | MyFun of ide * ide * exp * exp * exp
    | MyApply of ide * (exp option)

type evT = ...
    | MyClosure of ide * evT * exp * evT env

let rec eval e s =
    match e with ...
    | MyFun (f, x, edef, ebody, e) →
        let vdef = eval edef s in
        let closure = MyClosure (x, vdef, ebody, s) in
        let s1 = bind s f fclosure in
        eval e s1
    | MyApply (f, arg) → let closure = s f in
                (match closure with
                | MyClosure (x, vdef, ebody, s1) →
                    (match arg with
                    | Some e →
```

```
                let s2 = bind s1 x (eval arg s1) in
                eval ebody s2
            | None ->
                let s2 = bind s1 x vdef in
                eval ebody s2)
        | _ -> failwith "Error" );;
```

6)

$$\frac{T \triangleright cond1 \Rightarrow true \quad T \triangleright cond2 \Rightarrow true \quad T \triangleright e1 \Rightarrow v1}{T \triangleright on(cond1, e1, cond2, e2) \Rightarrow v1}$$

$$\frac{T \triangleright cond1 \Rightarrow true \quad T \triangleright cond2 \Rightarrow false \quad T \triangleright e1 \Rightarrow v1}{T \triangleright on(cond1, e1, cond2, e2) \Rightarrow v1}$$

$$\frac{T \triangleright cond1 \Rightarrow false \quad T \triangleright cond2 \Rightarrow true \quad T \triangleright e2 \Rightarrow v2}{T \triangleright on(cond1, e1, cond2, e2) \Rightarrow v2}$$

```
type exp = ...
    | on of exp * exp * exp * exp

let rec eval e s =
  match e with ...
  | on (cond1, e1, cond2, e2) ->
      let g1 = eval cond1 s in
      (match (typecheck (TBool, g1), g1) with
      | (true, Bool(true)) -> let g2 = eval cond2 s in
                              (match (typecheck (TBool, g2), g2) with
                              | (true, Bool(true)) -> eval e1 s
                              | (true, Bool(false)) -> eval e1 s
                              | (_, _) -> failwith "Type error")
```

```
| (true, Bool(false)) → let g2 = evd cond2 s in
                        (match (typecheck(TBool, g2), g2) with
                        | (true, Bool(true)) → evd e2 s
                        | (true, Bool(false)) → raise "False guards"
                        | (_, _) → failwith "type error"))
```

7)

```
type exp = ...
        | use of exp * exp * exp * exp

let rec evd e s =
    match e with ...
    | use (e1, e2, e3, cond) →
        let g = eval cond s in
        (match (typecheck(TBool, g), g) with
        | (true, Bool(true)) →
            let s1 = bind s x (eval e1 s) in
            eval e3 s1
        | (true, Bool(false)) →
            let s1 = bind s x (eval e2 s) in
            eval e3 s1
        | (_, _) → failwith "type error")
```

8)

$$T \triangleright Fu(x, e) \longrightarrow Fu\ Closure("x", e, T)$$

$$T \triangleright Var("Fu") \longrightarrow Fu\ Closure("x", body, T_{fDecl})$$

$$\frac{T \triangleright arg \Rightarrow va \quad T_{fDecl}[x = va] \triangleright body \Rightarrow V}{T \triangleright Fu\ Apply(Den("Fu"), arg) \Rightarrow V}$$

```
type exp = ...
        | Fu of ide * exp * exp list
        | Fu Apply of ide * exp

type evT = ...
        | Fu Closure of ide * exp * exp list * evT env

let rec eval e s =
    match e with
  | Fu (arg, fbody, lis) -> FuClosure (arg, fbody, lis, fDecEnv)
  | Apply (eF, eArg) ->
        let fclosure = eval eF s in
        (match fclosure with ...
        | FuClosure (arg, fbody, lis, fDecEnv) ->
              let aVal = eval eArg s in
                let rec search x lst =
                let typeName = getType x in
                match lst with
                | [] -> failwith "Elemento non presente"
                | y::l -> (match typecheck (typeName, y) with
                          | true -> if x=y then y else search x l
                          | false -> search x l )
```

```
        in
        let aenv = bind fDecEnv arg (List.map (fun x -> eval x s)
                                    lis) in
        eval fbody aenv
    | _ -> failwith "Error"
```

9) Posizione $p ::= 0, \ldots, 100$
Identificatori $I ::= \ldots$
Espressioni $e ::= I \mid p \mid moveLeft\ e \mid moveRight\ e \mid let\ I = e1\ in\ e2$

```
type ide = string

exception RunTimeError of string

type evT = ...
    | Unbound
    | Pos of int
    | String of string

type exp = ...
    | I of ide
    | Epos of int
    | MoveLeft of exp
    | MoveRight of exp
    | Let of ide * exp * exp

type 't env = ide -> 't     // Ambiente polimorfo

let emptyenv = function x -> UnBound     //Ambiente vuoto
```

```
let bind (s: evT env) (i: ide) (v: evT) =
    function (x: ide) -> if i = x then v else (s i)

let typecheck ((x,y): (tname * evT)) =
    match x with
    | TInt -> (match y with
              | Pos(n) -> true
              | _ -> raise (RunTime Exception "Wrong type"))
    | TString -> (match y with
              | String(s') -> true
              | _ -> raise (RunTime Exception "Wrong type"))

let mavel x =
    match (typecheck(Tint, x), x) with
    | (true, Pos(n)) -> if n=0 then Pos(0) else Pos(n-1)
    | _ -> failwith "Error"

let maveR x =
    match (typecheck(Tint, x), x) with
    | (true, Pos(n)) -> if n=100 then Pos(100) else Pos(n+1)
    | _ -> failwith "Error"

let rec eval e s =
    match e with
    | I(u) -> (s u)
    | EPos(u) -> Pos(u)
    | MoveLeft(e1) -> let p = eval e1 s in mavel p
    | MoveRight(e1) -> let p = eval e1 s in maveR p
    | Let(i, e1, body) -> eval body (bind s i (eval e1 s))
```