Mini Caml

```
// identificatori
type ide = string ;;

// tipi
type tname = TInt | TBool | TClosure | TRecClosure | TUnBound ;;

// Espressioni astratte
type exp =
    | EInt of int
    | CstTrue
    | CstFalse
    | EString of string
    | Den of ide
    // Operatori binari int → int
    | Sum of exp * exp
    | Diff of exp * exp
    | Prod of exp * exp
    | Div of exp * exp
    // Operatori int → bool
    | IsZero of exp
    | Eq of exp * exp
    | LessThan of exp * exp
    | GreaterThan of exp * exp
    // Operatori su bool
    | And of exp * exp
    | Or of exp * exp
    | Not of exp
```

```
// Controllo del flusso
| IfThenElse of exp * exp * exp
| Let of ide * exp * exp
| Letrec of ide * ide * exp * exp
| Fun of ide * exp
| Apply of exp * exp ;;
```

```
// Evaluation types
type evT =
    | Int of int
    | Bool of bool
    | String of string
    | Closure of ide * exp * evT env
    | RecClosure of ide * ide * exp * evT env
    | Unbound ;;
```

```
// Binding fra una stringa x e un valore primitivo v
let bind (s: evT env) (x: ide) (v: evT) =
    function (i: ide) -> if (i=x) then v else (s i) ;;
```

```
// Get type            // Associo ad ogni valore il suo descrittore di tipo
let getType (x: evT) : tname =
    match x with
    | Int (n) -> TInt
    | Bool (n) -> TBool
    | String (s) -> TString
    | Closure (i, e, en) -> TClosure
    | RecClosure (i, J, e, en) -> TRecClosure
    | Unbound -> TUnbound
```

```
// Type checking
let typecheck ((x,y): (tname * evT)) =
    match x with
    | TInt -> (match y with
               | Int(u) -> true
               | _ -> false )
    | TBool -> (match y with
               | Bool(u) -> true
              | _ -> false)
    | TString -> (match y with
               | String(u) -> true
               | _ -> false)
    | TClosure -> (match y with
               | Closure (i, e, en) -> true
               | _ -> false)
    | TRecClosure -> (match y with
                  | RecClosure (i, J, e, en) -> true
                  | _ -> false)
    | TUnbound -> (match y with
                  | Unbound -> true
                  | _ -> false) ;;

// Intereprete
let rec eval (e : exp) (s: evT env) : evT =
    match e with
    | EInt(n) -> Int(n)
    | CstTrue -> Bool(True)
    | Cst False -> Bool(false)
    | EString -> String (s)
    | Den(i) -> (s i)
| Prod (e1, e2) → int_times ((eval e1 s), (eval e2 s))
| Sum (e1, e2) → int_plus ((eval e1 s), (eval e2 s))
| Diff (e1, e2) → int_sub ((eval e1 s), (eval e2 s))
| Div (e1 , e2) → int_div ((eval e1 s), (eval e2 s))
| IsZero (e1) → is_zero (eval e1 s)
| Eq (e1, e2) → int_eq ((eval e1 s), (eval e2 s))
| LessThan (e1, e2) → less_than ((eval e1 s), (eval e2 s))
| GreaterThan (e1, e2) → greater_than ((eval e1 s), (eval e2 s))
| And (e1, e2) → bool_and ((eval e1 s), (eval e2 s))
| Or (e1, e2) → bool_or ((eval e1 s), (eval e2 s))
| Not (e1) → bool_not (eval e1 s)
| IfThenElse (e1, e2, e3) → let g = eval e1 s  in
                          (match (typecheck (TBool, g), g) with
                          | (true , Bool (true)) → eval e2 s
                          | (true , Bool (false)) → eval e3 s
                          | (_ , _) → raise (RuntimeError "Wrong type"))
| Let (i, e, ebody) → eval ebody (bind s i (eval e s))
| Fun (arg, ebody) → Closure (arg, ebody, s)
| LetRec (f, arg, fBody, letBody) →
     let benv = bind s f (RecClosure (f, arg, fBody, s)) in
     eval letBody benv
| Apply (eF, eArg) → let fclosure = eval eF s in
   (match fclosure with
    | Closure (arg, fbody, fDecEnv) → let aVal = eval eArg s in
                                     let aenv = bind fDecEnv arg aVal in
                                     eval fbody aenv
    | RecClosure (f, arg, fbody, fDecEnv) → let aVal = eval eArg s in
//estendo l'ambiente statico
                                     let rEnv = bind fDecEnv f fclosure in
//estendo rEnv con il passaggio
  dei parametri
                                     let aenv = bind rEnv arg aVal in
                                     eval fbody aenv
| _ → raise (RuntimeError "Wrong type")) ;;
```

Semantica operazionale

ifthenelse (Regole di inferenza)

$$\frac{\Sigma \triangleright cond \Rightarrow true \quad \Sigma \triangleright e_1 \Rightarrow v_1}{\Sigma \triangleright ifthenelse(cond, e_1, e_2) \Rightarrow v_1} \qquad \frac{\Sigma \triangleright cond \Rightarrow false \quad \Sigma \triangleright e_2 \Rightarrow v_2}{\Sigma \triangleright ifthenelse(cond, e_1, e_2) \Rightarrow v_2}$$

ifthenelse (Interpreete)

```
ifthenelse (cond , e1, e2) →
    let g = eval cond amb in
    match (typecheck ("bool", g), g) with
    | (true, Bool(true)) eval e1 amb
    | (true, Bool(false)) eval e2 amb
    | (_,_) → failwith "Non boolean guard"
```

Let (Regole di inferenza)

$$\frac{\Gamma \triangleright e_1 \Rightarrow v_1 \qquad \Gamma[x = v_1] \triangleright e_2 \Rightarrow v_2}{\Gamma \triangleright Let(x, e_1, e_2) \Rightarrow v_2}$$

Let (Interprete)

```
let rec eval ((e: exp), (amb: evT env)) =
    match e with . . .
    | Let (i, e, ebody) →
        eval ebody (bind amb i (eval e amb))
```

Function (Regole di inferenza)

Scoping statico

$$\left.\begin{array}{l} \Gamma \triangleright Fun(x, e) \Rightarrow Closure(\text{"}x\text{"}, e, \Gamma) \\[2em] \Gamma \triangleright Var(\text{"}f\text{"}) \Rightarrow Closure(\text{"}x\text{"}, body, \Gamma_{fDecl}) \end{array}\right\} \text{Astrazione}$$

$$\left.\dfrac{\Gamma \triangleright arg \Rightarrow va \quad \Gamma_{fDecl}[x = va] \triangleright body \Rightarrow v}{\Gamma \triangleright Apply(Den(\text{"}f\text{"}), arg) \Rightarrow v}\right\} \text{Applicazione}$$

$$\left.\dfrac{\Gamma[f = Closure(\text{"}x\text{"}, e, \Gamma)] \triangleright e' \Rightarrow v'}{\Gamma \triangleright Let(\text{"}f\text{"}, Fun(x, e), e') \Rightarrow v'}\right\} \text{Dichiarazione}$$

Scoping dinamico

$$\Gamma \triangleright Fun(\text{"}x\text{"}, e) \Rightarrow Funval(\text{"}x\text{"}, e) \; \} \; \text{Astrazione}$$

$$\left.\begin{array}{c} \Gamma \triangleright Den(\text{"}f\text{"}) \Rightarrow Funval(\text{"}x\text{"}, e) \\[2em] \dfrac{\Gamma \triangleright arg \Rightarrow va \quad \Gamma[x = va] \triangleright e \Rightarrow v}{\Gamma \triangleright Apply(Den(\text{"}f\text{"}), arg) \Rightarrow v} \end{array}\right\} \text{Applicazione}$$

Function (Interprete) Scoping statico

```
let rec eval ((e: exp) (amb: evT env)) =
    match e with ...
    | Fun (i, a) -> Closure (i, a, amb)
    | Apply (Den(f), eArg) ->
        let fclosure = lookup amb f in
        (match fclosure with
        | Closure (arg, fbody, fDecEnv) ->
            let aVal = eval eArg amb in
            let aenv = bind fDecEnv arg aVal in
                eval fbody aenv
        | _ -> failwith "Non functional value"
    | Apply (_, _) -> failwith "Application: not first order function" ;;
```

Scoping dinamico

```
type evT = Int of int
         | Bool of bool
         | Unbound
         | Funval of efun

and efun = ide * exp
```

```
let rec eval ((e: exp) (amb: evT env)) =
    match e with ...
    | Fun (arg, ebody) → Funval (arg, ebody)
    | Apply (Den(f), eArg) →
        let fval = lookup amb f in
        (match fval with
        | Funval (arg, fbody) →
            let aVal = eval eArg amb in
            let aenv = bind amb arg aVal in
                eval fbody aenv
        | _ → failwith "Non functional value"
    | Apply (_,_) → failwith "Application: not first order function";;
```