Es1: Si consideri la seguente grammatica, che definisce un semplice linguaggio funzionale

(Programma) Program Decl => Exp
(Dichiarazioni) Decl ::= ε | fun Ide(Ide) {Exp} ; Decl
(Identificatori) Ide ::= <definizione standard>
(Interi) Int ::= <definizione standard>
(Booleani) Bool ::= <definizione standard>
(Operatori) Op ::= + | - | * | = | <=
(Espressioni) Exp ::= Ide | Int | Bool
| Exp and Exp | Exp or Exp | not Exp
| Op(Exp, Exp)
| if Exp then Exp else Exp
| Ide(Exp)

Una dichiarazione di funzione ha la forma **fun f (x) { expr }** dove **f** è il nome della funzione, **x** il parametro formale, ed **expr** una espressione dove **x** può comparire libera.
Si noti che, come in C, le dichiarazioni di funzioni occorrono solo al "top-level" e definiscono quindi un ambiente globale nel quale deve essere valutata l'espressione finale del programma.
Si definisca una semantica operazionale mediante regole di inferenza che rispetti le seguenti specifiche

- l'applicazione funzionale deve essere valutata con riferimento all'ambiente globale determinato dalle dichiarazioni;
- tutte le altre operazioni hanno il significato visto a lezione.

Es2: Si verifichi la correttezza della semantica progettando ed eseguendo alcuni casi di test, sufficienti a testare tutti gli operatori.

- In particolare, si valuti il seguente programma, tenendo a mente che il risultato deve essere **5**

Program

- fun sub1 (n) { -(n, 1) };
- fun fib (n) { if =(n, 0) or =(n, 1) then n else +(fib(sub1(n)), fib (sub2(n))) };
- fun sub2 (m) { sub1(sub1(m)) }; ε

=>
fib (5)

Es3: Si definisca una sintassi astratta per il linguaggio, introducendo opportuni tipi di dati OCaml.

- Si definisca un interprete OCaml che corrisponda alla semantica introdotta in precedenza.
- Si traducano nella sintassi astratta proposta i casi di test proposti in precedenza, in particolare il programma indicato, e si valutino con l'interprete.

```
(* PRIMA VERSIONE *)
(* Espressioni *)

type ide = string ;;

type exp = EInt of int       | Den of ide          | App of ide * exp     | Add of exp * exp
        | Sub of exp * exp  | Mul of exp * exp  | Ifz of exp * exp * exp
        | Eq of exp * exp    | Leq of exp * exp  | Not of exp
        | And of exp * exp  | Or of exp * exp
;;

type def = Fun of ide * ide * exp;;

type prog = Prog of (def list) * exp;;

(* run time ci sono due possibilità per l'ambiente:
   1) un solo ambiente, che contiene sia funzioni globali che identificatori associati ad interi
(per semplicità non abbiamo booleani), introdotti per  passaggio di parametri.
   2) due ambienti separati, uno globale per funzioni, uno locale per identificatori.
Accettiamo quest'ultima:
nell'ambiente globale associo al nome di una funzione semplicemente una coppia: ide *
exp. L'ambiente globale e' gestito come se fosse scoping dinamico, quindi passato
all'invocazione, per garantire che contenga tutte le funzioni dichiarate. *)

exception EmptyEnv;;

(* Ambiente globale default *)
let fenv0 = fun x -> raise EmptyEnv ;;
(* oppure let fenv0 x = raise EmptyEnv ;; *)

(* Ambiente locale default *)
let env0 = fenv0;;

(* Estensione di ambiente *)
let ext env (x: string) v = fun y -> if x=y then v else env y ;;
(* let ext env (x: string)(v)(y: string) = if x = y then v else env y ;; *)

(*
env, fenv : 'a -> 'b
ext : ('a -> 'b) -> 'a -> 'b -> 'a -> 'b
env : string -> int
fenv : string -> (string * exp)
*)

(* valutazione di espressioni   NB:  booleani sono rappresentati da 0 (false) e 1 (true) *)
```

```
let rec eval (e: exp) env fenv = match e with
   | EInt i -> i
   | Den s -> env s
   | App(s,e2) -> let (par,body) = (fenv s) in (eval body (ext env par (eval e2 env fenv)) fenv)
   | Add (e1,e2) -> (eval e1 env fenv)+(eval e2 env fenv)
   | Sub (e1,e2) -> (eval e1 env fenv)-(eval e2 env fenv)
   | Mul (e1,e2) -> (eval e1 env fenv)*(eval e2 env fenv)
   | Not e1 -> if ((eval e1 env fenv) = 0)  then 1 else 0
   | Or (e1,e2) -> if (eval e1 env fenv) = 0 then (eval e2 env fenv) else 1
   | And (e1,e2) -> if (eval e1 env fenv) != 0 then (eval e2 env fenv) else 0
   | Eq (e1,e2) -> if ((eval e1 env fenv) = (eval e2 env fenv)) then 1 else 0
   | Leq (e1,e2) -> if ((eval e1 env fenv) <= (eval e2 env fenv)) then 1 else 0
   | Ifz (e1,e2,e3) -> if (eval e1 env fenv) = 1 then (eval e2 env fenv)
                           else (eval e3 env fenv);;

(* valutazione di dichiarazione: restituisce un ambiente globale *)
let rec dval (decls: def list) = match decls with
   |  [ ] -> fenv0
   |  Fun (fname, par, body)::rest -> ext (dval rest) fname (par, body);;

(* valutazione di programma: valuta l'espressione usando l'ambiente globale ottenuto dalle
dichiarazioni *)
let peval (p: prog) = match p with
      Prog (decls, expr) -> let fenv = dval(decls) in eval expr env0 fenv;;

(* ============================  TESTS  ================= *)

(* basico: no funzioni *)
let p1 = Prog ([ ], Add(EInt 4, EInt 5));;
peval p1;;

(* una funzione non ricorsiva: succ *)
let p2 = Prog ([Fun("succ", "x", Add(Den "x", EInt 1))], Add(App("succ", EInt 4), EInt 5));;
peval p2;;

(* funzione ricorsiva: triangolare *)
let p3 = Prog (
[Fun("tria", "x", Ifz(Eq(Den "x", EInt 0), EInt 5, Add(Den "x", App("tria", Sub(Den "x", EInt
1)))))], App("tria", EInt 4));;
peval p3;;

(* funzione ricorsiva: fattoriale *)
let p4 = Prog ([Fun("fact", "x", Ifz(Leq(Den "x", EInt 1), EInt 1, Mul(Den "x", App("fact",
Sub(Den "x", EInt 1)))))],
```

```
                        App("fact", EInt 3));;
peval p4;;

(* Programma test del progetto: Fibonacci che usa anche funzione definita dopo:
Program
fun sub1 (n) { -(n,1)};
fun fib (n) { if =(n,0) or = (n,1) then n else +( fib (sub1(n)), fib (sub2(n))) } ;
fun sub2 (m) { sub1(sub1(m)) }
=> fib (5)
*)

let ptest =
Prog
([Fun ("sub1", "n", Sub(Den "n", EInt 1));
  Fun ("fib", "n",
          Ifz(Or(Eq(Den "n", EInt 0), Eq(Den "n", EInt 1)), Den "n",
                Add(App("fib", App("sub1", Den "n")),
                    App("fib", App("sub2", Den "n"))))));
  Fun ("sub2", "m", Sub(Den "m", EInt 2))],
  App("fib", EInt 5));;

peval ptest;;    (* risultato: 5 *)


(* SECONDA VERSIONE *)

(* Ambiente polimorfo*)
type 't env = ide -> 't;;

(* Estensione di ambiente *)
let bind (r : 't env) (i : ide) (v : 't) = function x -> if x = i then v else r x;;
(* oppure let bind (r: 't env)(i: ide)(v: 't)(x: ide) = if x = i then v else r x ;; *)

exception EmptyEnv;;
exception WrongFun;;

(* Ambiente default *)
let env0 = fun x -> raise EmptyEnv ;;
(* oppure let env0 x = raise EmptyEnv ;; *)

(*
bind : ('a -> 'b) -> 'a -> 'b -> 'a -> 'b
venv = int env
fenv = (ide * exp) env
*)
```

```
(* runtime *)

(* valutazione di espressioni NB:  booleani sono rappresentati da 0 (false) e 1 (true) *)

let rec eval e venv denv = match e with
     | EInt i -> i
     | Den s -> venv s
     | App (s, e1) -> (match (denv s) with
          (par, body) -> let v = (eval e1 venv denv) in
                              let venv1 = (bind venv par v) in
                                let venv1 = (bind env0 par v) in (* cosi' abbiamo scoping statico *)
                              (* scoping dinamico con let venv1 = (bind venv par v) in *)
                                      eval body venv1 denv
                            | _ -> raise WrongFun)
     | Add (e1,e2) -> (eval e1 venv denv)+(eval e2 venv denv)
     | Sub (e1,e2) -> (eval e1 venv denv)-(eval e2 venv denv)
     | Mul (e1,e2) -> (eval e1 venv denv)*(eval e2 venv denv)
     | Not e1 -> if ((eval e1 venv denv) = 0)  then 1 else 0
     | Or (e1,e2) -> if (eval e1 venv denv) = 0 then (eval e2 venv denv) else 1
     | And (e1,e2) -> if (eval e1 venv denv) != 0 then (eval e2 venv denv) else 0
     | Eq (e1,e2) -> if ((eval e1 venv denv) = (eval e2 venv denv)) then 1 else 0
     | Leq (e1,e2) -> if ((eval e1 venv denv) <= (eval e2 venv denv)) then 1 else 0
     | Ifz (e1,e2,e3) -> if (eval e1 venv denv) = 1
                              then (eval e2 venv denv)
                              else (eval e3 venv denv)
;;

(* valutazione di dichiarazione: restituisce un ambiente globale *)
let rec dval (decls: def list) = match decls with
      | [ ] -> env0
      | Fun (fname, par, body)::rest -> bind (dval rest) fname (par, body);;

(* valutazione di programma: valuta l'espressione usando l'ambiente globale
ottenuto dalle dichiarazioni *)
let pval (p: prog) = match p with
     Prog(decls, exp) -> let fenv = (dval decls) in eval exp env0 fenv;;

(* ============================  TESTS  ================ *)

(* basico: no funzioni *)
let p1 = Prog([ ], Add(EInt 4, EInt 5));;
pval p1;;

(* una funzione non ricorsiva: succ *)
```

```
let p2 = Prog([Fun("succ", "x", Add(Den "x", EInt 1))], Add(App("succ", EInt 4), EInt 5));;
pval p2;;

(* funzione ricorsiva: triangolare *)
let p3 = Prog([Fun("tria", "x", Ifz(Eq(Den "x", EInt 0), EInt 5, Add(Den "x", App("tria",
Sub(Den "x", EInt 1)))))],
                App("tria", EInt 4));;
pval p3;;

(* funzione ricorsiva: fattoriale *)
let p4 = Prog([Fun("fact", "x", Ifz(Leq(Den "x", EInt 1), EInt 1, Mul(Den "x", App("fact",
Sub(Den "x", EInt 1)))))],
                App("fact", EInt 3));;

pval p4;;

(* Programma test del progetto: Fibonacci che usa anche funzione definita dopo:
Program
funz sub1 (n) { -(n,1)};
funz fib (n) { if =(n,0) or = (n,1) then n else +( fib (sub1(n)), fib (sub2(n))) } ;
funz sub2 (m) { sub1(sub1(m)) }
=> fib (4)
*)

let ptest =
  Prog([Fun("sub1", "n", Sub(Den "n", EInt 1));
        Fun("fib", "n",
            Ifz(Or(Eq(Den "n", EInt 0), Eq(Den "n", EInt 1)), Den "n",
                Add(App("fib", App("sub1", Den "n")),
                    App("fib", App("sub2", Den "n")))));
        Fun("sub2", "m", Sub(Den "m", EInt 2))],
       App("fib", EInt 4));;

pval ptest;;  (* risultato: 3 *)
```