Es1: Si consideri il semplice linguaggio funzionale con dichiarazioni globali visto nella esercitazione precedente e se ne estenda la sintassi in modo da includere la possibilità di passare il parametro alle funzioni sia per nome che per riferimento. In particolare, la sintassi conterrà almeno la seguente struttura di tipo
type param = Val of ide | Name of ide | Ref of ide ;;
type def = Var of ide * exp | Fun of ide * param * exp;;
Si definisca una semantica operazionale mediante regole di inferenza che rispetti le seguenti specifiche

nel passaggio per nome, l'espressione che rappresenta il parametro attuale della funzione chiamata deve essere valutata solo al momento nel quale il parametro formale è effettivamente acceduto, e l'ambiente di valutazione deve essere quello definito al momento nel quale la funzione è invocata;

nel passaggio per riferimento, il parametro attuale della funzione chiamata deve essere una variabile e deve conservare le eventuali modifiche anche dopo il termine della funzione;

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 i nuovi operatori introdotti.

Es3: Si definisca una sintassi astratta per il linguaggio, introducendo opportuni tipi di dati e un interprete OCaml che corrisponde alla semantica operazionale introdotta. Si traducano poi nella sintassi astratta proposta i casi di test proposti in precedenza, e si valutino con l'interprete.

```
(* Espressioni *)

type ide = string;;
type loc = int;;
type param = Val of ide | Name of ide | Ref of ide;;

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
;;

(* tipi ausiliari*)
type venv = ide -> evT and evT = Unbound | Loc of loc | ExpVal of exp * venv;;
type dvT = Undef | FunVal of param * exp * venv;;
type mvT = int;;
type denv = ide -> dvT;;
```

```
type store = loc -> mvT;;

type def = Var of ide * exp | Fun of ide * param * exp;;

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

(* eccezioni *)
exception EmptyEnv;;
exception WrongFun;;

(* Estensione di ambiente *)
let bind env lid v = function x -> if x = lid then v else env x;;
(* oppure let bind env lid v x = if x = lid then v else env x ;; *)
(* bind : ('a -> 'b) -> 'a -> 'b -> 'a -> 'b *)

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

(* Contatore di locazioni: e' un puntatore!! *)
(* la presenza di s e' concettuale: la newloc dovrebbe dipendere dallo store... *)
let count = ref 0;;
let newloc (s : store) = (incr count; !count);;

(* runtime *)

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

(*interprete*)
let rec lval (e : exp) (r : venv) (g : denv) (s : store) : evT = match e with
        | Den id -> r id
        | _ -> let tmp = (newloc s) in Loc tmp;;

let rec eval (e : exp) (r : venv) (g : denv) (s : store) : mvT = match e with
        | EInt i -> i
        | Den id -> let vClosure = (r id) in (match vClosure with
                | Loc lo -> s lo
                | ExpVal(e1, r1) -> eval e1 r1 g s
                | _ -> raise EmptyEnv)
        | App (f, arg) -> let fClosure = (g f) in (match fClosure with
                | FunVal(par, fBody, fDecEnv) -> auxval fClosure arg r g s
```

```
                | _ -> raise WrongFun)
        | Add (e1,e2) -> (eval e1 r g s)+(eval e2 r g s)
        | Sub (e1,e2) -> (eval e1 r g s)-(eval e2 r g s)
        | Mul (e1,e2) -> (eval e1 r g s)*(eval e2 r g s)
        | Not e1 -> if ((eval e1 r g s) = 0)  then 1 else 0
        | Or (e1,e2) -> if (eval e1 r g s) = 0 then (eval e2 r g s) else 1
        | And (e1,e2) -> if (eval e1 r g s) != 0 then (eval e2 r g s) else 0
        | Eq (e1,e2) -> if ((eval e1 r g s) = (eval e2 r g s)) then 1 else 0
        | Leq (e1,e2) -> if ((eval e1 r g s) <= (eval e2 r g s)) then 1 else 0
        | Ifz (e1,e2,e3) -> if (eval e1 r g s) = 1 then (eval e2 r g s)
                            else (eval e3 r g s)

(* valutazione di funzioni: restituisce un valore *)
and auxval (fClos : dvT) (arg : exp) (r : venv) (g : denv) (s : store) : mvT =
        match fClos with
                | FunVal(par, fBody, fDecEnv) ->  (match par with
                        | Val id -> let tmp = (newloc s) in
                                let v = (eval arg r g s) in
                                eval fBody (bind fDecEnv id (Loc tmp)) g (bind s tmp v)
                        | Name id -> let tmp =  (ExpVal(arg, r)) in
                                eval fBody (bind fDecEnv id tmp) g s
                        | (* si passa l'ambiente al momento della chiamata di funzione *)
                         Ref id -> let tmp = (lval arg r g s) in (match tmp with
                                | Loc lo -> eval fBody (bind fDecEnv id tmp) g s
                                | (* se e' una locazione nuova, lo e' indefinito in s *)
                                 _ -> failwith("RefPar error") ) )
                | _ -> raise WrongFun

(* valutazione di dichiarazioni: restituisce un ambiente globale *)
and dval decls (r : venv) (g : denv) (s : store) : venv * denv * store  =
        match decls with
                | [ ] -> (r, g, s)
                | Fun(fname, par, body)::rest ->
                        let g1 = (bind g  fname (FunVal(par, body, r))) in dval rest r g1 s
                | Var(vid, e1)::rest -> let v = (eval e1 r g s) in
                        let tmp = (newloc s) in
                        let r1 = (bind r vid (Loc tmp)) in
                        let s1 = (bind s tmp v) in
                        dval rest r1 g s1;;

(*    Fun("sub1", Val "n", Sub(Den "n", Den "base1")) *)
```

```
(* valutazione di programma: valuta l'espressione usando l'ambiente globale
ottenuto dalle dichiarazioni *)
let pval p = match p with
   Prog(decls, e) ->
     let (venv, denv, store) = (dval decls env0 env0 env0) in (eval e venv denv store);;
(* ============================  TESTS  ================ *)

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

(* una funzione non ricorsiva: succ *)
let p2 = Prog([Fun("succ", Val "x", Add(Den "x", EInt 1))], Add(App("succ", EInt 4),
EInt 5));;
pval p2;;
(* restituisce - : mvT = 10, lo stesso con Name "x", mentre con Ref "x" avrebbe
lanciato Exception: EmptyEnv  *)

(* funzione ricorsiva: triangolare *)
let p3 = Prog([Fun("tria", Val "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([Var("test", EInt 3); Fun("fact", Val "x", Ifz(Leq(Den "x", EInt 1), EInt 1,
Mul(Den "x", App("fact", Sub(Den "x", EInt 1))))],
                          App("fact", Den "test"));;
pval p4;;

(* Programma test del progetto: Fibonacci che usa anche funzione definita dopo:
Program
      var base0 { 0 }; va base1 { 1 };
      fun sub1 (m) { m - base1 };
      fun fib (n) { if =(n, base0) or =(n, base1) then n else +(fib(sub1(n)), fib (sub2(n)))
};
      fun sub2 (m) { sub1(sub1(m)) };
      var add { 3 };
      var test { base1 + add };
      var add { 5 } &epsilon;
*)
```

```
let ptest =
Prog
([Var("base0", EInt 0); Var("base1", EInt 1);
  Fun("sub1", Val "n", Sub(Den "n", Den "base1"));
  Fun("fib", Val "n",
          Ifz(Or(Eq(Den "n", Den "base0"), Eq(Den "n", Den "base1")), Den "n",
               Add(App("fib", App("sub1", Den "n")),
                    App("fib", App("sub2", Den "n")))));
  Fun("sub2", Val "m", Sub(Sub(Den "m", Den "base1"), Den "base1"));
  Fun ("inc", Ref "n", Add(Den "n", Den "base1"));
  Var("base", EInt 3);
  Var("test", Add(App("inc", Den "base"), Den "base"))
],
  App("fib", Den "test"));;
pval ptest;; (* risultato:  13 *)
```

Es4: Si consideri adesso il semplice linguaggio funzionale visto a lezione, si escludano gli operatori ricorsivi e se ne estenda la sintassi in modo da includere la possibilità di passare il parametro alle funzioni sia per nome che per riferimento. In particolare, la sintassi conterrà almeno la seguente struttura di tipo

```
type param = Val of ide | Name of ide | Ref of ide ;;
type exp = ... | Fun of param * exp ;;
```

Es5: Si verifichi la correttezza della semantica progettando ed eseguendo alcuni casi di test, sufficienti a testare i nuovi operatori introdotti.

Es6: Si definisca una sintassi astratta per il linguaggio, introducendo opportuni tipi di dati e un interprete OCaml che corrisponde alla semantica operazionale introdotta. Si traducano poi nella sintassi astratta proposta i casi di test proposti in precedenza, e si valutino con l'interprete.

```
type ide = string;;
type loc = int;;
type param = Val of ide | Name of ide | Ref of ide
type exp =
        | Eint of int
        | Ebool of bool
        | Den 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
        | FunCall of exp * exp
        | Fun of param * exp ;;

(*ambienti polimorfi*)
type 't env = ide -> 't;;
type 't store = loc -> 't;;
let emptyenv (v : 't) = function x -> v;;
let applyenv (r : 't env) (i : ide) = r i;;
let applystore (s : 't store) (l : loc) = s l;;
let bindE (r : 't env) (id : ide) (v : 't) = function x -> if x = id then v else applyenv r x;;
let bindM (s : 't store) (l : loc) (v : 't) = function x -> if x = l then v else applystore s x;;

(*tipi memorizzabili ed esprimibili*)
type mvT = Int of int | Bool of bool | Undef;;
type evT = Loc of loc | Unbound | FunVal of evFun | ExpVal of evExp
        and evFun = param * exp * evT env
        and evExp = exp * evT env;;

(*rts*)
(*type checking*)
let typecheck (s : string) (v : mvT) : bool = match s with
        | "int" -> (match v with
                | Int(_) -> true
                | _ -> false)
        | "bool" -> (match v with
                | Bool(_) -> true
                | _ -> false)
        | _ -> failwith("not a valid type");;

(*funzioni primitive*)
let prod x y = if (typecheck "int" x) && (typecheck "int" y)
                        then (match (x,y) with (Int(n),Int(u)) -> Int(n*u))
                else failwith("Type error");;
```

```
let sum x y = if (typecheck "int" x) && (typecheck "int" y)
                    then (match (x,y) with (Int(n),Int(u)) -> Int(n+u))
              else failwith("Type error");;

let diff x y = if (typecheck "int" x) && (typecheck "int" y)
                    then (match (x,y) with (Int(n),Int(u)) -> Int(n-u))
              else failwith("Type error");;

let eq x y = if (typecheck "int" x) && (typecheck "int" y)
                    then (match (x,y) with (Int(n),Int(u)) -> Bool(n=u))
              else failwith("Type error");;

let minus x = if (typecheck "int" x)
                    then (match x with Int(n) -> Int(-n))
              else failwith("Type error");;

let iszero x = if (typecheck "int" x)
                    then (match x with Int(n) -> Bool(n=0))
              else failwith("Type error");;

let vel x y = if (typecheck "bool" x) && (typecheck "bool" y)
                    then (match (x,y) with (Bool(b),Bool(e)) -> (Bool(b||e)))
              else failwith("Type error");;

let et x y = if (typecheck "bool" x) && (typecheck "bool" y)
                    then (match (x,y) with (Bool(b),Bool(e)) -> Bool(b&&e))
              else failwith("Type error");;

let non x = if (typecheck "bool" x)
                    then (match x with
                              | Bool(true) -> Bool(false)
                              | Bool(false) -> Bool(true))
              else failwith("Type error");;

(*contatore di locazioni: e' un puntatore!!*)
(* la presenza di s e' concettuale: la newloc dovrebbe dipendere dallo store... *)
let count = ref 0;;
let newloc (s : mvT store) = (incr count; !count);;

(*ambienti default*)
let env0 = (emptyenv Unbound);;
let store0 = (emptyenv Undef);;
```

```
(*interprete*)
let rec dval (e : exp) (r : evT env)  (s : mvT store) : evT = match e with
        | Fun(p, a) -> FunVal(p, a, r)
        | Den id -> (applyenv r id)
        |  _ -> Loc (newloc s);;

let rec eval (e : exp) (r : evT env) (s : mvT store) : mvT = match e with
        | Eint n -> Int n
        | Ebool b -> Bool b
        | Den id -> let tmp = (dval e r s) in (match tmp with
                | Loc lo -> applystore s lo
                | ExpVal(e1, r1) -> eval e1 r1 s
                | _ -> failwith("DenExp error"))
        | IsZero a -> iszero (eval a r s)
        | Eq(a, b) -> eq (eval a r s) (eval b r s)
        | Prod(a, b) -> prod (eval a r s) (eval b r s)
        | Sum(a, b) -> sum (eval a r s) (eval b r s)
        | Diff(a, b) -> diff (eval a r s) (eval b r s)
        | Minus a -> minus (eval a r s)
        | And(a, b) -> et (eval a r s) (eval b r s)
        | Or(a, b) -> vel (eval a r s) (eval b r s)
        | Not a -> non (eval a r s)
        | Ifthenelse(a, b, c) -> let g = (eval a r s) in
                if (typecheck "bool" g)
                        then (if g = Bool(true) then (eval b r s) else (eval c r s))
                else failwith ("Guard error")
        | Let(id, e1, e2) -> let tmp = (dval e1 r s) in (match  tmp with
                | FunVal(p, a, r) -> eval e2 (bindE r id tmp) s
                | ExpVal(e3, r1) -> let tmp = (dval e3 r1 s) in (match tmp with
                        | Loc lo -> eval e2 (bindE r id tmp) (bindM s lo (eval e3 r1 s))
                        |  _ -> failwith("ExpVal error"))
                | Loc lo -> let r1 = (bindE r id tmp) in
                        let s1 = (bindM s lo (eval e1 r s)) in eval e2 r1 s1
                |_ -> failwith("LetExp error"))
        | FunCall(id, arg) -> let fClosure = (dval id r s) in (match fClosure with
                | FunVal(par, fBody, fDecEnv) -> auxval fClosure arg r s
                | _ -> failwith("FunCallExp error"))
and auxval (e : evT) (arg : exp) (r : evT env) (s : mvT store) : mvT = match e with
        | FunVal(par, fBody, fDecEnv) -> (match par with
                | Val id ->  let tmp = (newloc s) in
                  eval fBody (bindE fDecEnv id (Loc  tmp)) (bindM s tmp (eval arg r s))
```

```
                | Name id -> let tmp =  (ExpVal(arg, r)) in
                  eval fBody (bindE fDecEnv id tmp) s
                | (* si passa l'ambiente al momento della chiamata di funzione *)
                 Ref id -> let tmp = (dval arg r s) in (match tmp with
                        | Loc lo -> eval fBody (bindE fDecEnv id tmp) s
                        | (* se e' una locazione nuova, lo e' indefinito in s *)
                          _ -> failwith("RefPar error") ) )
        |_ -> failwith("FunCalltExp error");;

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

(* basico: no let *)
let env0 = emptyenv Unbound;;

let e1 = FunCall(Fun(Ref "y", Sum(Den "y", Eint 1)), Eint 3);;
eval e1 env0;;

let e2 = FunCall(Let("x", Eint 2, Fun(Ref "y", Sum(Den "y", Den "x"))), Eint 3);;
eval e2 env0;;

(*Non torna *)
let e2 = Let("x",Eint 2,
                  "f", Fun(Ref "y", FunCall(Fun(Ref "x", Let("x", Eint 2, Fun(Ref "y",
Sum(Den "y", Den "x"))), Eint 3))));;

eval e2 env0;;
```