Es1: Si consideri il semplice linguaggio funzionale con dichiarazioni globali e memoria visto nella esercitazione 6bis e se ne modifichi la semantica in modo tale che se una funzione passa il proprio parametro formale per riferimento, alla fine della chiamata il valore calcolato del corpo della funzione viene assegnato al parametro attuale.
Ad esempio, il seguente programma deve restituire il valore 9.

```
Prog(
        [ Var("base", EInt 1);
          Fun("iden", Val "n", Den "n");
          Fun("inc", Ref "n", Add(Den "n", EInt 1));
          Var("test", Add(App("inc", Den "base"), Den "base"))
        ],
        App("iden", Add(Den "test", App("inc", Den "test")))
)
```

Si ridefinisca la semantica operazionale vista in precedenza in modo che rispetti le nuove condizioni.

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

Es3: Si ridefinisca 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;;

(*interprete*)
(* valutazione di assegnamenti: restituisce uno store *)
let rec cval (e : exp) (r : venv) (g : denv) (s : store) : (mvT * store) = match e with
        | EInt i -> (i, s)
        | Den id -> let vClosure = (r id) in (match vClosure with
                | Loc lo -> (s lo, s)
                | ExpVal(e1, r1) -> cval e1 r1 g s
                | _ -> raise EmptyEnv)
        | App (f, arg) -> let fClosure = (g f) in (match fClosure with
```

```
                | FunVal(par, fBody, fDecEnv) -> auxcval fClosure arg r g s
                | _ -> raise WrongFun)
        | Add (e1,e2) -> let (v1, s1) = (cval e1 r g s) in
                let (v2, s2) = (cval e2 r g s1) in (v1+v2, s2)
        | Sub (e1,e2) -> let (v1, s1) = (cval e1 r g s) in
                let (v2, s2) = (cval e2 r g s1) in (v1-v2, s2)
        | Mul (e1,e2) -> let (v1, s1) = (cval e1 r g s) in
                let (v2, s2) = (cval e2 r g s1) in (v1*v2, s2)
        | Not e1 -> let (v1, s1) = (cval e1 r g s) in if (v1 = 0)  then (1, s1) else (0, s1)
        | Or (e1,e2) -> let (v1, s1) = (cval e1 r g s) in
                if (v1 = 0) then (cval e2 r g s1) else (1, s1)
        | And (e1,e2) -> let (v1, s1) = (cval e1 r g s) in
                if (v1 != 0) then (cval e2 r g s1) else (0, s1)
        | Eq (e1,e2) -> let (v1, s1) = (cval e1 r g s) in
                let (v2, s2) = (cval e2 r g s1) in if (v1 = v2) then (1, s2) else (0, s2)
        | Leq (e1,e2) -> let (v1, s1) = (cval e1 r g s) in
                let (v2, s2) = (cval e2 r g s1) in if (v1 <= v2) then (1, s2) else (0, s2)
        | Ifz (e1,e2,e3) -> let (v1, s1) = (cval e1 r g s) in
                if v1 = 1 then (cval e2 r g s1)
                else (cval e3 r g s1)
and auxcval (fClos : dvT) (arg : exp) (r : venv) (g : denv) (s : store) : (mvT * store) =
  match fClos with
        | FunVal(par, fBody, fDecEnv) ->  (match par with
                | Val id -> let tmp = (newloc s) in
                        let (v1, s1) = (cval arg r g s) in
                        cval fBody (bind fDecEnv id (Loc tmp)) g (bind s1 tmp v1)
                | Name id -> let tmp =  (ExpVal(arg, r)) in
                        cval 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 -> let (v1, s1) =
                (cval fBody (bind fDecEnv id tmp) g s) in (v1, (bind s1 lo v1))
                        | (* se e' una locazione nuova, lo e' indefinito in s *)
                         _ -> failwith("RefPar error") ) )
        | _ -> raise WrongFun
;;

(* valutazione di dichiarazioni: restituisce un ambiente globale *)
let rec 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, vbody)::rest -> let (v, s1) = (cval vbody r g s) in
            let tmp = (newloc s1) in
            let r1 = (bind r vid (Loc tmp)) in
            let s2 = (bind s1 tmp v) in
            dval rest r1 g s2;;


(* 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
        let (v1, s1) = (cval e venv denv store) in v1;;

(* ============================  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;;

(*                        Val id -> let tmp = (newloc s) in
                                    let (v1, s1) = (cval arg r g s) in cval fBody (bind
fDecEnv id (Loc tmp)) g (bind s1 tmp v1) |
  *)

(* 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 }; var 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 ptest0 =
Prog
([  Var("base", EInt 1);
    Fun("iden", Val "n", Den "n");
    Fun ("inc", Ref "n", Add(Den "n", EInt 1));
   Var("test", Add(App("inc", Den "base"), Den "base"))
 ],
 App("iden", Add(Den "test", App("inc", Den "test"))));;

pval ptest0;; (* risultato: 9 *)

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: 21 *)
```