PROGRAMMAZIONE II
AA 2012-13
Prima Valutazione Intermedia

Es1. Si consideri un frammento del linguaggio funzionale didattico la cui sintassi astratta sia definita nel modo seguente

```
type exp =
  | Eint of int
  | Den of ide
  | Sum of exp * exp
  | Quot of exp * exp
  | Let of ide * exp * exp
```

Supponiamo di estendere il linguaggio con opportuni meccanismi per trattare le eccezioni che si possono manifestare durante l'esecuzione di un programma. In particolare, supponiamo di avere due tipi di eccezioni,

```
type exc = VU | DZ (* VU = Variable Unbound, DV = Division By Zero *)
```

Quando si verifica una eccezione viene interrotta la normale esecuzione del programma. Quindi un programma puo' terminare in maniera corretta oppure puo' terminare a causa di una situazione anomala generata da una eccezione. Per esempio il programma

```
Let ("x",Eint 2, Quot (Sum(Den "x",Eint 3), Eint 0))
```

non termina correttamente: viene sollevata l'eccezione "Division by Zero". Supponiamo di estendere il linguaggio con una primitiva linguistica per catturale e gestire le eccezioni. Sintatticamente scriviamo **try**(exp)**with** excepion-list. L'espressioni all'interno del blocco **try** puo' sollevare una eccezione, la lista all'interno del blocco **with** descrive le operazioni che devono essere eseguite per trattare opportunamente l'eccezione eventualmente sollevata durante la valutazione dell'espressione. Ad esempio, l'espressione,

```
Let ("x",Eint 2,
           Try(Quot ( Sum(Den "x",Eint 3), Eint 0),
           [(DZ, Den "x");(VU, Eint 1)]))
```

quando viene sollevata l'eccezione "Division by Zero" esegue l'espressione che restituisce il valore della variabile `x`. In sintesi, i passi che determinano il comportamento del costrutto **try - with -** sono:

- Si esegue l'espressione all'interno del blocco **try**
- Se l'esecuzione non genera situazioni anomale si restituisce il valore calcolato
- Se l'esecuzione dell'espressione solleva una eccezione, si esegue l'espressione corrispondente all'eccezione come descritto nel blocco **with**

Assumendo le definizioni seguenti

```
type exc = VU | DZ;;

type exp =
  | Eint of int
  | Den of ide
  | Sum of exp * exp
  | Quot of exp * exp
  | Let of ide * exp * exp
  | Try of exp * ((exc * exp) list) ;;

type env = (string*eval) list;;
```

Si specifichi l'interprete ricorsivo del frammenti del linguaggio funzionale esteso con le eccezioni.

Es2. Si consideri il seguente programma funzionale

```
let eq m n = n = m in
 let rec sumTo n = if n = 0 then 0 else n + sumTo(n-1) in
 let choose m n = if less m n then n else sumTo(m) in
 choose 3 4 ;;
```

- Tradurre il programma nel linguaggio didattico,
- Assumendo scoping statico, tradurre ogni uso degli identificatori con la coppia (numero di passi da effettuare sulla catena statica, posizione relativa).

Traccia Soluzione

Es1

```
type eval =
        | Int of int
        | Bool of bool
        | VarUnbound
        | DivByZero;;

type ide = string;;

(* Operations on Exception *)
type exc = VU | DZ

type exp =
  | Eint of int
  | Den of ide
  | Sum of exp * exp
  | Quot of exp * exp
  | Let of ide * exp * exp
  | Try of exp * ((exc * exp) list) ;;

type env = (string*eval) list;;

(* Operations on Eval *)

   let typecheck (x, y) =
      match x with
      | "int" ->
          (match y with
          |  Int(u) -> true
          | _ -> false)
      | _ -> failwith ("not a valid type") ;;

    let plus (x,y) =
      if typecheck("int",x) & typecheck("int",y)
      then
           (match (x,y) with
           | (Int(u), Int(w)) -> Int(u+w)
        | _ -> failwith("error"))
      else failwith ("type error") ;;

let rec checkEx(exl, ex) =
  match ex with
    | VarUnbound ->
    (match exl with
      |  [] -> false
      | (VU, e)::exls -> true
```

```
      | (DZ, e) ::exls -> checkEx(exls, ex))
    | DivByZero ->
    (match exl with
      |  [] -> false
      | (DZ, e)::exls -> true
      | (VU, e)::exls -> checkEx(exls, ex))
    | _ -> failwith("error");;

let rec recoveryEx(exl, ex) =
  match ex with
    | VarUnbound ->
    (match exl with
      |  [] -> failwith("error")
      | (VU, e)::exls -> e
      | (DZ, e) ::exls -> recoveryEx(exls, ex))
    | DivByZero ->
    (match exl with
      |  [] -> failwith("error")
      | (DZ, e)::exls -> e
      | (VU, e)::exls -> recoveryEx(exls, ex))
    | _ -> failwith("error") ;;

let rec lookup env x  =
    match env with
    | []        -> VarUnbound
    | (y, v)::r -> if x=y then v else lookup r x;;

let rec sem ((e:exp),(r:env) ) =
  match e with
    | Eint(n) -> Int(n)
    | Den(i) -> lookup r i
    | Sum(a,b) ->  let arg1 = sem(a, r) in
                       if arg1 = VarUnbound then VarUnbound
                       else let arg2 = sem(b,r) in
                            if arg2 = VarUnbound then VarUnbound
                            else plus(arg1, arg2)
     | Quot(a,b) ->  let arg1 = sem(a, r) in
                       (match arg1 with
       | VarUnbound -> VarUnbound
       | Int(p) -> let arg2 = sem(b,r) in
                            (match arg2 with
                              | VarUnbound -> VarUnbound
                              | Int(0) -> DivByZero
                              | Int(w) ->if (typecheck("int", arg1)&&typecheck("int",arg2))
                                                                then Int(p mod w)
                                                                else failwith("error")
      | _ -> failwith("error"))
                       | _ -> failwith("error"))
    | Let(i,e1,e2) -> let ival = sem(e1, r) in
      let env1 = (i, ival) :: r in
```

```
      sem(e2, env1)
   | Try(e, ecc) ->
     let ev = sem(e, r) in
    ( match ev with
     |  VarUnbound  ->  if checkEx(ecc, VarUnbound) then sem(recoveryEx(ecc,VarUnbound), r)
                                                    else failwith("eccezione non catturata")
     |  DivByZero  ->  if checkEx(ecc, DivByZero) then sem(recoveryEx(ecc,DivByZero),r)
                                                    else failwith("eccezione non catturata")
     |  Int(n) -> Int(n)
     |  _ -> failwith("error"));;
```

Es2.

```
(*
let eq m n = n= m in
  let rec sumTo n = if n = 0 then 0 else n + sumTo(n-1) in
  let choose m n = if eq m n then n else sumTo(m) in
  choose 3 4 ;;
*)
let es = Let
            ("eq",Fun (["m"; "n"],Eq(Den "n",Den "m")),
             Let
               ("sumTo",
                Rec
                  ("sumTo",
                   Fun
                     (["n"],
                      Ifthenelse
                        (Iszero (Den "n"),Eint 0,
                         Sum(Den"n",Appl (Den "sumTo",[Diff (Den "n",Eint 1)])))),
                Let
                  ("choose",
                   Fun
                     (["m"; "n"],
                      Ifthenelse
                        (Appl (Den "eq",[Den "m"; Den "n"]),Den "n",
                         Appl (Den "sumTo",[Den "m"]))),Appl (Den "choose",[Eint 3 ; Eint 4])))));;

let staticexp =
  SBind
    (SFun (Sequ (Access (0,1),Access (0,0))),
     SBind
       (SRec
          (SFun
             (Sifthenelse
                (Siszero (Access (0,0)),SInt 0,
                 Splus
                   (Access (0,0),
                    SAppl (Access (1,0),[Sdiff (Access (0,0),SInt 1)])))),
```

```
SBind
  (SFun
     (Sifthenelse
        (SAppl (Access (2,0),[Access (0,0); Access (0,1)]),
         Access (0,1),SAppl (Access (1,0),[Access (0,0)]))),
   SAppl (Access (0,0),[SInt 3; SInt 4])))
```