

```

(* syntax *)

type ide = string

type boolean =
  | True
  | False

type exp =
  | Eint of int
  | Eplus of (exp * exp)
  | Eminus of (exp * exp)
  | Eide of ide
  | Ebool of boolean
  | Eeql of (exp * exp)
  | Eleg of (exp * exp)
  | Enot of exp
  | Eand of (exp * exp)
  | Eor of (exp * exp)
  | Eifthenelse of (exp * exp * exp)
  | Eapp of exp * exp
  | Efun of ide * exp
  | Elet of (ide * exp * exp)

type com =
  | Cassign of ide * exp
  | Cvar of ide * exp
  | Cconst of ide * exp
  | Cifthenelse of exp * pseq * pseq
  | Cwhile of exp * pseq
  | CdoNTimes of exp * pseq
  | Cforall of ide * pseq * (exp*exp)

and pseq =
  | Pseq of com * pseq
  | Pend

type prog = Prog of pseq * exp

let rec exp_to_string (e: exp) =
  match e with
  | Eint i -> sprintf "%d" i
  | Eplus (e1, e2) -> sprintf "(%s + %s)" (exp_to_string e1)
(exp_to_string e2)
  | Eminus (e1, e2) -> sprintf "(%s - %s)" (exp_to_string e1)
(exp_to_string e2)
  | Eide i -> i
  | Ebool b ->
    match b with
    | True -> "true"
    | False -> "false"
  | Enot e -> sprintf "(not %s)" (exp_to_string e)
  | Eand (e1, e2) -> sprintf "(%s and %s)" (exp_to_string e1)
(exp_to_string e2)

```

```

    | Eor (e1, e2) -> sprintf "(%s or %s)" (exp_to_string e1)
(exp_to_string e2)
    | Eeql (e1, e2) -> sprintf "(%s == %s)" (exp_to_string e1)
(exp_to_string e2)
    | Eleq (e1, e2) -> sprintf "(%s <= %s)" (exp_to_string e1)
(exp_to_string e2)
    | Eifthenelse (c, e1, e2) ->
        sprintf "if %s then (%s) else (%s)" (exp_to_string c)
(exp_to_string e1) (exp_to_string e2)
    | Efun (arg, body) -> sprintf "fun %s -> %s" arg (exp_to_string
body)
    | Eapp (f, arg) -> sprintf "%s(%s)" (exp_to_string f)
(exp_to_string arg)
    | Elet (v, e1, e2) -> sprintf "(let %s = %s in %s)" v
(exp_to_string e1) (exp_to_string e2)

```

```

let rec com_to_string (c: com) =
    match c with
    | Cassign (i, e) -> sprintf "%s := %s" i (exp_to_string e)
    | Cvar (v, e) -> sprintf "var %s := %s" v (exp_to_string e)
    | Cconst (v, e) -> sprintf "const %s = %s" v (exp_to_string e)
    | Cifthenelse (cond, cthen, celse) ->
        sprintf "if %s\nthen %s\nelse %s" (exp_to_string cond)
(pseq_to_string cthen) (pseq_to_string celse)
    | Cwhile (cond, body) -> sprintf "while %s\n%s" (exp_to_string
cond) (pseq_to_string body)
    | CdoNTimes (cond, body) -> sprintf "do %s
times\n%s" (exp_to_string cond) (pseq_to_string body)
    | Cforall (ide, pseq, list) -> sprintf "For %s in %A do:\n%s" ide
list (pseq_to_string pseq)

```

```

and pseq_to_string (s: pseq) =
    match s with
    | Pseq (c, Pend) -> sprintf "%s" (com_to_string c)
    | Pseq (c, q) -> sprintf "%s;\n%s" (com_to_string c)
(pseq_to_string q)
    | Pend -> ""

```

```

let prog_to_string (p: prog) =
    match p with
    | Prog (s, e) -> sprintf "%s;\nreturn %s" (pseq_to_string s)
(exp_to_string e)

```

(\* error handling \*)

```

let unbound_identifier_error ide =
    failwith (sprintf "unbound identifier %s" ide)

```

```

let negative_natural_number_error () =
    failwith "natural numbers must be positive or zero"

```

```

let type_error () = failwith "type error"

```

```

let memory_error () =

```

```

    failwith "access to a location that is not available"

let not_a_location_error i =
    failwith (sprintf "not a location: %s" i)

(* semantic domains *)

type eval =
  | Int of int
  | Bool of bool
  | Fun of (ide * env * exp)

and loc = int

and mval = eval

and store = int * (loc -> mval) (* il primo elemento della coppia è
la minima locazione non definita *)

and dval =
  | E of eval
  | L of loc

and env = ide -> dval

let empty_store = (0, (fun l -> memory_error ()))

let apply_store st l = (snd st) l

let allocate: store -> loc * store =
  fun st ->
    let l = fst st in
    let l1 = l + 1 in
    let st1 = (l1, snd st) in
    (l, st1)

let update: store -> loc -> mval -> store =
  fun st l mv ->
    match st with
    | (maxloc, fn) -> let fn1 l1 = if l = l1 then mv else fn l1
in (maxloc, fn1)

let empty_env = fun v -> unbound_identifier_error v

let bind e v r = fun v1 -> if v1 = v then r else e v1

let apply_env e v = e v

let rec eval_to_string (e: eval) =
  match e with
  | Int i -> sprintf "%d" i
  | Bool b -> if b then "true" else "false"
  | Fun _ -> "function"

```

(\* denotational semantics \*)

```
let rec esem: exp -> env -> store -> eval =
  fun e ev st ->
    match e with
    | Eint i ->
      if i < 0 then
        negative_natural_number_error ()
      else
        Int i
    | Eplus (e1, e2) ->
      (let s1 = esem e1 ev st in
       let s2 = esem e2 ev st in

       match (s1, s2) with
       | (Int i1, Int i2) -> Int(i1 + i2)
       | _ -> type_error ())
    | Eminus (e1, e2) ->
      let s1 = esem e1 ev st in
      let s2 = esem e2 ev st in

      (match (s1, s2) with
       | (Int i1, Int i2) ->
         if i1 >= i2 then
           Int(i1 - i2)
         else
           negative_natural_number_error ()
       | _ -> type_error ())
    | Ebool b ->
      (match b with
       | True -> Bool true
       | False -> Bool false)
    | Eeql (e1, e2) ->
      let s1 = esem e1 ev st in
      let s2 = esem e2 ev st in

      (match (s1, s2) with
       | (Int i1, Int i2) ->
         if i1 = i2 then
           Bool true
         else
           Bool false
       | _ -> type_error ())
    | Eleq (e1, e2) ->
      let s1 = esem e1 ev st in
      let s2 = esem e2 ev st in

      (match (s1, s2) with
       | (Int i1, Int i2) ->
         if i1 <= i2 then
           Bool true
         else
           Bool false
```

```

    | _ -> type_error ()
| Eand (e1, e2) ->
    let s1 = esem e1 ev st in
    let s2 = esem e2 ev st in

    (match (s1, s2) with
    | (Bool b1, Bool b2) -> Bool(b1 && b2)
    | _ -> type_error ())
| Eor (e1, e2) ->
    let s1 = esem e1 ev st in
    let s2 = esem e2 ev st in

    (match (s1, s2) with
    | (Bool b1, Bool b2) -> Bool(b1 || b2)
    | _ -> type_error ())
| Enot e ->
    let s = esem e ev st in

    (match s with
    | Bool b -> Bool(not b)
    | _ -> type_error ())
| Eifthenelse (c, e1, e2) ->
    let sc = esem c ev st in

    (match sc with
    | Bool b -> esem (if b then e1 else e2) ev st
    | _ -> type_error ())
| Eide i ->
    let value = apply_env ev i

    match value with
    | L l -> apply_store st l
    | E e -> e
| Elet (v, e1, e2) ->
    let s1 = esem e1 ev st in
    let ev1 = bind ev v (E s1) // NOTE: s1 is an "eval"
    esem e2 ev1 st
| Efun (arg, body) -> Fun(arg, ev, body)
| Eapp (f, arg) -> // NB: f is an _expression_, not an
identifier
    let fn = esem f ev st in

    match fn with
    | Fun (par, ev1, body) ->
        let s = esem arg ev st
        esem body (bind ev1 par (E s)) st // Se scoping
dinamico, al posto di "ev1" c'è "ev"
    | _ -> type_error ()

let rec csem: com -> env -> store -> (env * store) =
    fun c ev st ->
        match c with
        | Cassign (i, e) ->
            let s = esem e ev st

```

```

    match apply_env ev i with
    | L l -> let st1 = update st l s in (ev, st1)
    | _ -> not_a_location_error i
| Cvar (i, e) ->
    let s = esem e ev st in
    let (newloc, st1) = allocate st in
    let st2 = update st1 newloc s in
    let ev1 = bind ev i (L newloc) in
    (ev1, st2)
| Cconst (i, e) ->
    let s = esem e ev st in
    let ev1 = bind ev i (E s) in
    (ev1, st)
| Cifthenelse (cond, cthen, celse) ->
    let s = esem cond ev st in

    match s with
    | Bool b ->
        if b then
            pssem cthen ev st // ERRORE: questo causa
scoping dinamico, vale a dire, le variabili dichiarate nel ramo then
possono essere viste dal seguito del programma
        else
            pssem celse ev st
    | _ -> type_error ()
| Cwhile (cond, body) ->

    let rec aux ev st =
        let cresult = esem cond ev st in

        match cresult with
        | Bool b ->
            if not b then
                (ev, st)
            else
                match st with
                | (newloc, _) -> let (_, (_, stfn')) = pssem
body ev st in aux ev (newloc, stfn')
                | _ -> type_error ()

        aux ev st // inizio da ambiente e stato di chiamata

| CdoNTimes (expr, body) ->
    // NON TESTATA!!! Fatta per esercizio
    let rec aux n ev (newloc, stfn) =
        if n <= 0 then
            (ev, (newloc, stfn))
        else
            let (ev1, (_, stfn')) = pssem body ev (newloc,
stfn)
            aux (n - 1) ev (newloc, stfn')

    let cresult = esem expr ev st

```

```

        match cresult with
        | Int i -> aux i ev st
        | _ -> type_error ()

    | CForAll (ide, pseq, (e1,e2)) ->
        match (esem e1 ev st, esem e2 ev st) with
        | (Int i1,Int i2) ->
            let (ev1, st1) = pssem pseq (bind ev ide (E (Int
i1))) st
            let (ev2, st2) = pssem pseq (bind ev ide (E (Int
i2))) st1
            (ev,st2)
        | _ -> type_error()

```

```

and pssem: pseq -> env -> store -> (env * store) =
  fun s ev st ->
    match s with
    | Pend -> (ev, st)
    | Pseq (c, q) ->
        match csem c ev st with
        | (ev1, st1) -> pssem q ev1 st1

```

```

let rec psem: prog -> env -> store -> eval =
  fun p ev st ->
    match p with
    | Prog (s, e) ->
        let (ev1, st1) = pssem s ev st
        esem e ev1 st1

```

(\* test \*)

```

let eval: prog -> unit =
  fun p ->
    printf "\n%s \n\n==> " (prog_to_string p)

    try
      printf "%s\n" (eval_to_string (psem p empty_env
empty_store))
    with
      | Failure message -> printfn "error: %s\n" message

```

```

let a () = failwith ""

```

```

let l =
  [ Prog(
    Pseq(
      Cvar("x", Eint 0),
      Pseq(
        Cvar("y", Eint 0),
        Pseq(CForAll("x", Pseq(Cassign("y", Eplus(Eide
"y", Eide "x"))), Pend), (Eint 1,Eint 2))), Pend)
    )
  ),

```

```
    ) ]   Eide "y"
```

```
let main = List.iter eval l
```

```
(*
```

```
> dotnet run
```

```
var x := 98;  
var y := 28;  
while (not (x == y))  
  if (x <= y)  
  then y := (y - x)  
  else x := (x - y);  
return x
```

```
==> 14
```

```
*)
```