JOHANNES KEPLER
UNIVERSITÄT LINZ

Formal Semantics of Programming Languages (SS 2025) – Assignment B1:

Student Name: Lyna Hocine.

Student Number: 12449045.

Email Address: lyna.hocine@etu.sorbonne-universite.fr.

1. Introduction :

This assignment B1 extends the previous assignment A by:

- Adapting the denotational semantics presented in Figures 7.2, 7.3, 7.6 and 7.20 to fit our specific language.
- Implementing the denotational semantics in Ocaml as a function that :
  1. Takes a list of initial parameter values.
  2. Returns a list of final parameter values after program execution
  3. May raise exceptions for not well-defined expressions (e.g., division by zero).

2. Key changes from Assignment A to B :

- **Procedure parameters:** In assignment A, procedures only supported input parameters, while assignment B extends the language to accommodate both input and output parameters, enabling procedures to return values through state updates.
- **State model:** The state model in assignment A was a simple mapping from variable names to values. In contrast, assignment B introduces a more sophisticated state-handling mechanism that supports the dynamic update of variables, especially output parameters, in line with denotational semantics.
- **Procedure environment**: the procedure environment initially relied on a basic list-based structure, which has been updated in assignment b with a more precise functional mapping (**proc_env**).
- **Type checker:** Assignment A provided a simple basic type checker for expressions and commands while assignment B includes an extension that implements variable-to-address translation.

3. Code explanation :

Our Ocaml **eval_command** function matches the functional version in **Fig. 7.2**:

Specifically, the eval_command function mirrors the structure of this formalism: variable updates are handled through the **update_state** function, which reflects the state transformation defined for assignments.
Sequencing of commands is implemented through recursive calls to **eval_command**, preserving the evaluation order specified in the semantics.
Conditional constructs are evaluated by pattern matching on the Boolean values produced by expression evaluation, directly corresponding to the semantics of if-then-else in **Fig 7.2**.
Loop constructs, such as the while loop, are implemented using a recursive helper function.

The denotational semantics of expressions in **Fig 7.6** define a simple language where expressions evaluate to integers, with the semantic domain restricted to natural numbers. In assignment A, the implementation supports only integer expressions. In assignment B, the expression language extended to incorporate Boolean expressions, so the semantic domain expanded to include both integers and Booleans, expressed as **Value = N U B**. Therefore, **eval_expr** is therefore adapted to handle both numerical and Boolean values appropriately.

The denotational semantics presented in **Figure 7.20**, which introduces procedures, is central to the extensions implemented in Assignment B. The interpreter adapts the notion of a procedure environment **proc_env** to align with this **ProcCall** construct in our implementation follows the denotational principles described in the figure, where the evaluation of a procedure call involves creating an extended state for the procedure body, executing the body, and subsequently updating the caller's state. Specifically, input parameters are first evaluated in the current state and output parameters are allocated in the extended state with initial values (set to 0). After executing the procedure body, the final values of the output parameters are extracted from procedure's state and written back to the corresponding variables in the caller's state.

4. **Explanation of the Optional part :**
Figure 7.22 introduces the concept of translating variables to addresses through an environment mapping **Env: Var -> Addr.** Each declared variable is assigned a unique address, and the environment is extended with each new declaration. The Ocaml implementation reflects this directly through an environment function of type **string -> address**, where a new address is generated for each variable declaration. The environment is updated with the helper function **update_env**, ensuring that the most recent binding is always correctly represented.
Figure 7.23 defines the denotational semantics with variables translated to addresses, introducing a store, which maps addresses to values: **store: address -> value**.
The store is extended and updated during both variable declarations and assignments. The Ocaml code is implements this through a store function and an **update_store** operation that modifies the values at a specific address. During expression evaluation, the interpreter looks up a variable's address using the environment and retrieves its value from the store.

This address-based model is integrated throughout the interpreter. Variables are bound to addresses via the environment, matching the translation mechanism in **Fig 7.22**, while values are stored and updated using a store structure, as defined in **Fig 7.23**.
Fresh addresses are generated incrementally using a counter, ensuring uniqueness and consistency. The evaluation of expressions **eval_expr** relies on resolving addresses to values through the store. Procedure calls extend the environment and store appropriately, allocating addresses for both input and output parameters in alignment with semantics of **Fig 7.23**.

This guarantees correct memory management for both global and local variables, prevents address conflicts, and ensures that procedures manage their own scoped environments as expected.

5. **How to use the code for the Assignment and the optional part :**

The core part of the assignment is implemented using functions like **eval_expr, eval_command** and **run_program**. These allow us to:

- Define programs with global variables, procedures, and a main procedure.
- Support arithmetic and Boolean expressions, conditionals, loops and procedure calls with input and output parameters.
- Run the program by calling **run_program** with our program and a list of initial parameter values.
- Get the final values of the program parameters as output.

For the optional part, an alternative version of the interpreter introduces addresses and explicit memory model. In this version of code:

- Each variable is assigned a unique address.
- The environment maps variable names to addresses.
- The store keeps track of values at those addresses.
- Expressions and commands are evaluated using both the environment and store.
- Procedure calls allocate fresh addresses for parameters, ensuring proper memory isolation.

How to use it:

- This version typically provides a modified **eval_expr, eval_command and run_program** that operate with the address-based model.
- We run the programs in a similar way to the core version, but the interpreter now tracks addresses and memory explicitly.
- The output remains the same in terms of parameter values.

6. **Descriptions of convincing test runs of the programs :**

To demonstrate the correctness and functionality of the code, I used **example 7.7,** the Euclidean gcd program. This program follows this logic:

- The div procedure computes the quotient and remainder via subtractions.
- The gcd procedure repeatedly applied div, reducing the larger of two numbers until one reaches zero.
- The result is stored in the output parameter g.
- The sum of the quotients computed during the process is stored in the output parameter n.

Example parameter call:

let result = run_program program_example [IntVal 30; IntVal 18; IntVal 0; IntVal 0];;

Expected output:

a = 30, b = 18, gcd = 6, sum_quotients = 4.

The interpreter also includes robust error handling:

- Division by zero triggers an **EvalError**.
- Incorrect types in expressions, such as adding a Boolean to an integer, raise an **EvalError**.
- Mismatched procedure arguments result in a runtime error.

7. **Source code in a structured and easy to read way :**

```
(*Type def *)
type value =
  | IntVal of int
  | BoolVal of bool

type state = string -> value
exception EvalError of string

     (*Expressions: variables, arithmetic operations, logical
operations *)
```

```
type expr =
  | Int of int
  | Bool of bool
  | Var of string
  | Add of expr * expr
  | Sub of expr * expr
  | Mult of expr * expr
  | Div of expr * expr
  | Neg of expr
  | Eq of expr * expr
  | Lt of expr * expr
  | Leq of expr * expr
  | Gt of expr * expr
  | Geq of expr * expr
  | And of expr * expr
  | Or of expr * expr
  | Not of expr

    (*Sort to specify types of variables and parameters *)

type sort =
  |IntSort
  |BoolSort

type parameter = string * sort (* function parameters are name
and sort pairs *)

                    (*Commands of the language *)
type command =
  | VarDeclar of string * sort * expr (*variable declaration *)
  | Assign of string * expr (* Assignment *)
  | Seq of command * command (*Sequencing of commands *)
  | IfThenElse of expr * command * command (*If-Then-Else
conditional *)
  | IfThen of expr * command (*If-Then conditional *)
  | While of expr * command (*While loop *)
  | ProcCall of string * expr list (* Procedure call *)
  | ProcDecl of string * parameter list *parameter list * command
* command (* Procedure declaration *)

                (*Declarations of global variables and procedures *)
type declaration =
  | GlobalVar of string * sort
  | Procedure of string * parameter list * parameter list *
command

                 (*A complete program: declarations, main procedure
name, parameters and the body *)

type program = Program of declaration list * string * parameter
list * command

                         (*Evaluating expressions under the current
state*)

let rec eval_expr (e: expr) (s : state) : value =
  match e with
  | Int n -> IntVal n
  | Bool b -> BoolVal b
  | Var x -> s x
  | Add (e1, e2) -> (match eval_expr e1 s, eval_expr e2 s with
     | IntVal v1, IntVal v2 -> IntVal (v1 + v2)
     | _ -> raise (EvalError "Addition requires integers"))
  | Sub (e1, e2) -> (match eval_expr e1 s, eval_expr e2 s with
     | IntVal v1, IntVal v2 -> IntVal (v1 - v2)
     | _ -> raise (EvalError "Subtraction requires integers"))
  | Mult (e1, e2) -> (match eval_expr e1 s, eval_expr e2 s with
     | IntVal v1, IntVal v2 -> IntVal (v1 * v2)
     | _ -> raise (EvalError "Multiplication requires integers"))
  | Div (e1, e2) -> (match eval_expr e1 s, eval_expr e2 s with
     | IntVal v1, IntVal v2 -> if v2 <> 0 then IntVal (v1 / v2)
         else raise (EvalError "Division by zero")
     | _ -> raise (EvalError "Division requires integers"))
  | Neg e1 -> (match eval_expr e1 s with
     | IntVal v -> IntVal (-v)
     | _ -> raise (EvalError "Negation requires integer"))
  | Eq (e1, e2) -> (match eval_expr e1 s, eval_expr e2 s with
     | IntVal v1, IntVal v2 -> BoolVal (v1 = v2)
     | BoolVal b1, BoolVal b2 -> BoolVal (b1 = b2)
     | _ -> raise (EvalError "Equality requires same type"))
  | Lt (e1, e2) -> (match eval_expr e1 s, eval_expr e2 s with
     | IntVal v1, IntVal v2 -> BoolVal (v1 < v2)
     | _ -> raise (EvalError "Less-than requires integers"))
  | Leq (e1, e2) -> (match eval_expr e1 s, eval_expr e2 s with
     | IntVal v1, IntVal v2 -> BoolVal (v1 <= v2)
     | _ -> raise (EvalError "Less-equal requires integers"))
  | Gt (e1, e2) -> (match eval_expr e1 s, eval_expr e2 s with
     | IntVal v1, IntVal v2 -> BoolVal (v1 > v2)
     | _ -> raise (EvalError "Greater-than requires integers"))
  | Geq (e1, e2) -> (match eval_expr e1 s, eval_expr e2 s with
     | IntVal v1, IntVal v2 -> BoolVal (v1 >= v2)
     | _ -> raise (EvalError "Greater-equal requires integers"))
  | And (e1, e2) -> (match eval_expr e1 s, eval_expr e2 s with
     | BoolVal b1, BoolVal b2 -> BoolVal (b1 && b2)
     | _ -> raise (EvalError "And requires Booleans"))
  | Or (e1, e2) -> (match eval_expr e1 s, eval_expr e2 s with
     | BoolVal b1, BoolVal b2 -> BoolVal (b1 || b2)
     | _ -> raise (EvalError "Or requires Booleans"))
  | Not e1 -> (match eval_expr e1 s with
     | BoolVal b -> BoolVal (not b)
     | _ -> raise (EvalError "Not requires Boolean"))

             (* Extends a state with parameter bindings *)
```

```
let rec extend_state (params : parameter list) (vals : value list)
(outer : state) : state =
  match params, vals with
  | [], [] -> outer
  | (name, _) :: ps, v :: vs ->
      let rest = extend_state ps vs outer in
      fun x -> if x = name then v else rest x
  | _ -> raise (EvalError "Mismatched parameter and value
count")

          (*Updates a state with new variable assignment *)

let update_state (s : state) (x : string) (v : value) : state =
  fun y -> if y = x then v else s y
      (*A proc_def is a list of parameters and a command body *)
type proc_def  = parameter list * parameter list * command
                 (*A proc_env maps the name to the procedure
definition *)
type proc_env = string -> proc_def
    (* An empty procedure environment *)
let empty_proc_env : proc_env = fun _ -> raise(EvalError
"Undefined Procedure")

    (*Extending a  procedure environment with a new binding *)

let extend_proc_env (proc: string) (inputs: parameter list)
(outputs: parameter list) (body: command) (penv: proc_env):
proc_env =
  fun name -> if name = proc then (inputs,outputs, body) else
penv name

      (* Evaluating commands under a state and procedure
environment *)
```

```
let rec eval_command (cmd : command) (s : state) (penv :
proc_env)  : state =
  match cmd with
  | VarDeclar (x, _, e) -> update_state s x (eval_expr e s)
  | Assign (x, e) -> update_state s x (eval_expr e s)
  | Seq (c1, c2) -> eval_command c2 (eval_command c1 s penv)
penv
  | IfThenElse (e, c1, c2) ->
      (match eval_expr e s with
       | BoolVal true -> eval_command c1 s penv
       | BoolVal false -> eval_command c2 s penv
       | _ -> raise (EvalError "Condition must be boolean"))
  | IfThen (e, c1) ->
      (match eval_expr e s with
       | BoolVal true -> eval_command c1 s penv
       | BoolVal false -> s
       | _ -> raise (EvalError "Condition must be boolean"))
  | While (e, c1) ->
      let rec loop st =
        match eval_expr e st with
        | BoolVal true -> loop (eval_command c1 st penv)
        | BoolVal false -> st
        | _ -> raise (EvalError "Condition must be boolean")
      in loop s

  | ProcDecl (name, inputs, outputs, body, cont) ->
      let new_env = extend_proc_env name inputs outputs body
penv in
      eval_command cont s new_env

  | ProcCall (name, args) ->
      let (inputs, outputs, body) = penv name in
      if List.length args <> List.length inputs + List.length outputs
then
        raise (EvalError "Mismatched parameter and value count ")
      else
        let input_args, output_args =
          let rec split n l =
            if n = 0 then ([], l)
            else match l with
              |x::xs ->
                  let (first, rest) =split (n -1) xs in
                  (x::first, rest)
              |[] -> raise (EvalError "not enough arguments ")
          in
          split (List.length inputs) args
        in
        let input_vals = List.map (fun e -> eval_expr e s)
input_args in
        let proc_state = extend_state inputs input_vals s in
        let proc_state_with_outputs = extend_state outputs
(List.map (fun _ ->IntVal 0) outputs) proc_state in
        let final_state = eval_command body
proc_state_with_outputs penv in
        let updated_state =
          List.fold_left2 (fun st out_param out_arg_expr ->
              let out_arg_name = match out_arg_expr with
                | Var name -> name
                | _ -> raise (EvalError "Output arguments must be
variables")
              in
              update_state st out_arg_name (final_state (fst
out_param))
            ) s outputs output_args
        in
        updated_state

        (*Building a procedure environment from top-level
declarations *)
let build_proc_env (decls : declaration list): proc_env =
  let rec helper ds penv =
    match ds with
    | [] -> penv
    |Procedure( name,inputs, outputs, body) :: rest ->
        helper rest (extend_proc_env name inputs outputs body
penv)
    |_ :: rest -> helper rest penv
  in helper decls empty_proc_env

    (*Running the full program given input values for parameters
*)
let run_program (Program (decls, main_name, params, body))
(input_vals : value list) : value list =
  let default = fun _ -> IntVal 0 in
  let globals = List.fold_left (fun acc d -> match d with
      |GlobalVar (name, _) -> (name, IntSort)::acc
      |_ -> acc) [] decls
  in
  let state_with_globals = extend_state globals (List.init
(List.length globals) (fun _ -> IntVal 0)) default in
  let full_state = extend_state params input_vals
state_with_globals in
  let proc_env = build_proc_env decls in
  let final_state = eval_command (ProcCall (main_name,
List.map (fun (name,_) -> Var name) params))
      full_state proc_env in List.map(fun(name,_) -> final_state
name) params

    (*Example 7.7 *)
let program_example = Program ([
    GlobalVar("c",IntSort);
    Procedure("div", [("a",IntSort);("b",IntSort)],
[("q",IntSort);("r",IntSort)],
              Seq(Assign("q",Int 0),
                  Seq(Assign("r",Var "a"),
                    While(Geq(Var "r",Var "b"),
                        Seq(Assign("q",Add(Var "q",Int 1)),
Assign("r",Sub(Var "r",Var "b")))))));
  Procedure("gcd", [("a_in",IntSort);("b_in",IntSort)],
[("g",IntSort);("n",IntSort)],
          Seq(VarDeclar("a",IntSort,Var "a_in"),
             Seq(VarDeclar("b",IntSort,Var "b_in"),
                Seq(VarDeclar("q",IntSort,Int 0),
                   Seq(VarDeclar("r",IntSort,Int 0),
                      Seq(Assign("c",Int 0),
                         Seq(While(And(Gt(Var "a",Int 0),Gt(Var
"b",Int 0)),
                                IfThenElse(Geq(Var "a",Var "b"),
                                        Seq(ProcCall("div",[Var
"a";Var "b";Var "q";Var "r"]),
                                           Seq(Assign("a",Var "r"),
Assign("c",Add(Var "c",Var "q")))),
                                        Seq(ProcCall("div",[Var
"b";Var "a";Var "q";Var "r"]),
                                           Seq(Assign("b",Var "r"),
Assign("c",Add(Var "c",Var "q")))))),
                            Seq(IfThenElse(Gt(Var "a",Int 0),
Assign("g",Var "a"), Assign("g",Var "b")),
                               Assign("n",Var "c")))))))));
  Procedure("main", [("a",IntSort);("b",IntSort)],
[("c",IntSort);("d",IntSort)],
          ProcCall("gcd", [Var "a"; Var "b"; Var "c"; Var "d"]))
 ], "main",
[("a",IntSort);("b",IntSort);("c",IntSort);("d",IntSort)],
  Seq(Assign("dummy",Int 0), Assign("dummy",Int 0)))

(* Example run *)
let result = run_program program_example [IntVal 30; IntVal
18; IntVal 0; IntVal 0];;
```

```
match result with [IntVal a; IntVal b; IntVal c; IntVal d] ->
Printf.printf "a=%d, b=%d, gcd=%d, sum_quotients=%d\n" a b c
d | _ -> Printf.printf "Unexpected output\n";;
```

The optional part code :

```
(*Type def *)
type value =  (*values in the language are integers and booleans
*)
  | IntVal of int
  | BoolVal of bool

type address = int

type env = string -> address
type store = address -> value

exception EvalError of string (* for eval errors *)

     (*Expressions : variables, arithmetic operations, logical
operations *)
type expr =
  | Int of int
  | Bool of bool
  | Var of string
  | Add of expr * expr
  | Sub of expr * expr
  | Mult of expr * expr
  | Div of expr * expr
  | Neg of expr
  | Eq of expr * expr
  | Lt of expr * expr
  | Leq of expr * expr
  | Gt of expr * expr
  | Geq of expr * expr
  | And of expr * expr
  | Or of expr * expr
  | Not of expr

type sort = (*Sort to specify types of variables and parameters *)
  |IntSort
  |BoolSort

type parameter = string * sort (* function parameters are name
and sort pairs *)

              (*Commands of the language *)
type command =
  | VarDeclar of string * sort * expr (*variable declaration *)
  | Assign of string * expr (* Assignment *)
  | Seq of command * command (*Sequencing of commands *)
  | IfThenElse of expr * command * command (*If-Then-Else
conditional *)
  | IfThen of expr * command (*If-Then conditional *)
  | While of expr * command (*While loop *)
  | ProcCall of string * expr list (* Procedure call *)
  | ProcDecl of string * parameter list * command * command (*
Procedure declaration *)

              (*Declarations of global variables and procedures *)
type declaration =
  | GlobalVar of string * sort
  | Procedure of string * parameter list * parameter list *
command

              (*A complete program: declarations, main procedure
name, parameters and the body *)
type program = Program of declaration list * string * parameter
list * command
```

```
let update_env ( env : env) (x: string ) (a : address) : env =
  fun y -> if y = x then a else env y

let update_store (store: store) (a: address) (v: value): store =
  fun x -> if x = a then v else store x

let empty_env : env = fun _ -> raise( EvalError "Unbound
variable ")
let empty_store : store = fun _ -> raise( EvalError "Uninitialized
memory ")

                        (*Evaluationg expressions  under the current
state*)

let rec eval_expr (e : expr) (env : env) (store : store) : value =
  match e with
  | Int n -> IntVal n
  | Bool b -> BoolVal b
  | Var x -> store (env x)
  | Add (e1, e2) -> (match eval_expr e1 env store , eval_expr e2
env store with
      | IntVal v1, IntVal v2 -> IntVal (v1 + v2)
      | _ -> raise (EvalError "Addition requires integers"))
  | Sub (e1, e2) -> (match eval_expr e1 env store, eval_expr e2
env store with
      | IntVal v1, IntVal v2 -> IntVal (v1 - v2)
      | _ -> raise (EvalError "Subtraction requires integers"))
  | Mult (e1, e2) -> (match eval_expr e1 env store, eval_expr e2
env store with
      | IntVal v1, IntVal v2 -> IntVal (v1 * v2)
      | _ -> raise (EvalError "Multiplication requires integers"))
  | Div (e1, e2) -> (match eval_expr e1 env store, eval_expr e2
env store with
      | IntVal v1, IntVal v2 -> if v2 <> 0 then IntVal (v1 / v2)
```

```
        else raise (EvalError "Division by zero")
    | _ -> raise (EvalError "Division requires integers"))
  | Neg e1 -> (match eval_expr e1 env store with
    | IntVal v -> IntVal (-v)
    | _ -> raise (EvalError "Negation requires integer"))
  | Eq (e1, e2) -> (match eval_expr e1 env store, eval_expr e2
env store with
    | IntVal v1, IntVal v2 -> BoolVal (v1 = v2)
    | BoolVal b1, BoolVal b2 -> BoolVal (b1 = b2)
    | _ -> raise (EvalError "Equality requires same type"))
  | Lt (e1, e2) -> (match eval_expr e1 env store, eval_expr e2 env
store with
    | IntVal v1, IntVal v2 -> BoolVal (v1 < v2)
    | _ -> raise (EvalError "Less-than requires integers"))
  | Leq (e1, e2) -> (match eval_expr e1 env store, eval_expr e2
env store with
    | IntVal v1, IntVal v2 -> BoolVal (v1 <= v2)
    | _ -> raise (EvalError "Less-equal requires integers"))
  | Gt (e1, e2) -> (match eval_expr e1 env store, eval_expr e2 env
store with
    | IntVal v1, IntVal v2 -> BoolVal (v1 > v2)
    | _ -> raise (EvalError "Greater-than requires integers"))
  | Geq (e1, e2) -> (match eval_expr e1 env store, eval_expr e2
env store with
    | IntVal v1, IntVal v2 -> BoolVal (v1 >= v2)
    | _ -> raise (EvalError "Greater-equal requires integers"))
  | And (e1, e2) -> (match eval_expr e1 env store, eval_expr e2
env store with
    | BoolVal b1, BoolVal b2 -> BoolVal (b1 && b2)
    | _ -> raise (EvalError "And requires booleans"))
  | Or (e1, e2) -> (match eval_expr e1 env store, eval_expr e2 env
store with
    | BoolVal b1, BoolVal b2 -> BoolVal (b1 || b2)
    | _ -> raise (EvalError "Or requires booleans"))
  | Not e1 -> (match eval_expr e1 env store with
      | BoolVal b -> BoolVal (not b)
      | _ -> raise (EvalError "Not requires boolean"))


        (*A proc_def is a list of parameters and a command body *)
type proc_def  = parameter list * command
                  (*A proc_env maps the name to the procedure
definition *)
type proc_env = string -> proc_def
   (* An empty procedure environment *)
let empty_proc_env : proc_env = fun _ -> raise(EvalError
"Undefined Procedure")

   (*Extending a  procedure environment with a new binding *)
let extend_proc_env (proc: string) (params: parameter list)
(body: command) (penv: proc_env): proc_env =
  fun name -> if name = proc then (params, body) else penv
name

        (* Evaluating commands under a state and procedure
environment *)
let rec eval_command (cmd : command) (env : env) (store :
store) (penv : proc_env) (next_addr : address)  : env * store *
address =
  match cmd with
  | VarDeclar (x, _, e) -> let v = eval_expr e env store in
      let store' = update_store store next_addr v in
      let env' = update_env env x next_addr in
      (env', store', next_addr + 1)

  | Assign (x, e) -> let a = env x in
      let v = eval_expr e env store in
      let store' = update_store store a v in
      (env, store', next_addr)
  | Seq (c1, c2) -> let(env1, store1, addr1) = eval_command c1
env store penv next_addr in
      eval_command c2 env1 store1 penv addr1

  | IfThenElse (e, c1, c2) ->
      (match eval_expr e env store with
       | BoolVal true -> eval_command c1 env store penv
next_addr
       | BoolVal false -> eval_command c2 env store penv
next_addr
       | _ -> raise (EvalError "Condition must be boolean"))

  | IfThen (e, c1) ->
      (match eval_expr e env store with
       | BoolVal true -> eval_command c1 env store penv
next_addr
       | BoolVal false -> (env, store, next_addr)
       | _ -> raise (EvalError "Condition must be boolean"))

  | While (e, c1) ->
      let rec loop env store addr =
        match eval_expr e env store  with
        | BoolVal true -> let (env', store', addr') = eval_command c1
env store penv addr in
            loop env' store' addr'
        | BoolVal false -> (env, store, addr)
        | _ -> raise (EvalError "Condition must be boolean")
      in loop env store next_addr

  | ProcDecl (name, params, body, cont) ->
      let new_penv = extend_proc_env name params body penv in
      eval_command cont env store new_penv next_addr

  | ProcCall (name, args) ->
      let (formals, body) = penv name in
      let arg_vals = List.map( fun e -> eval_expr e env store) args
in
      let (env_proc, store_proc, addr_after) =
        List.fold_left2 (fun (env_acc, store_acc, addr_acc)
(pname,_) v ->
            let env_acc' = update_env env_acc pname addr_acc in
            let store_acc' = update_store store_acc addr_acc v in
            (env_acc', store_acc', addr_acc+1))
          (empty_env, store, next_addr) formals arg_vals
      in let (_, store_final, addr_final) = eval_command body
env_proc store_proc penv addr_after in
      (env, store_final, addr_final)

        (*Building a procedure environment from top-level
declarations *)
let build_proc_env decls  =
  let rec aux decls penv  =
    match decls with
    | [] -> penv
    |Procedure( name, formals,_, body) :: rest ->
        aux rest (extend_proc_env name formals body penv)
    |_ :: rest -> aux rest penv
  in aux decls empty_proc_env

    (*Running the full program given input values for parameters
*)
let run_program (Program (decls, _, params, body)) (input_vals :
value list) : value list =
  let penv = build_proc_env decls in
  let (env, store, next_addr) =
    List.fold_left2 (fun(env, store, addr) (x, _) v ->
        let env' = update_env env x addr in
        let store' = update_store store addr v in
        (env', store', addr+1))
      (empty_env, empty_store, 0)
      params input_vals
    in
    let (env_final, store_final, _) = eval_command body env store
  penv next_addr in
    List.map(fun (x, _) -> store_final(env_final x)) params


     (*Example 7.7 *)
  let gcd_program =
    Program (
     [ (* Global variable *)
       GlobalVar ("c", IntSort);

       (* Procedure div(a, b; ref q, r) *)
       Procedure( "div", [("a", IntSort); ("b", IntSort)],
                 [("q", IntSort); ("r", IntSort)],

                 Seq(
                   Assign ("q", Int 0),
                   Seq (
                     Assign ("r", Var "a"),
                     Seq (
                       While (Geq (Var "r", Var "b"),
                              Seq (
                                Assign ("q", Add (Var "q", Int 1)),
                                Assign ("r", Sub (Var "r", Var "b"))
                              )
                             ),
                       Assign ("c", Add(Var "c", Int 1))
                     )
                   )
                 )
                );
       (* Procedure gcd(a, b; ref g, n) *)
```

```
   Procedure ("gcd", [("a", IntSort); ("b", IntSort)], [("g",
IntSort); ("n", IntSort)],
           Seq (
            Assign ("c", Int 0),
            Seq (
             While (And (Gt (Var "a", Int 0), Gt (Var "b", Int
0)),
                 Seq (
                  VarDeclar ("c", IntSort, Int 0),
                  IfThenElse (
                   Geq(Var "a", Var "b"),
                   ProcCall("div", [Var "a"; Var "b"; Var "c";
Var "a"]),
                   ProcCall ("div", [Var "b"; Var "a"; Var "c";
Var "b"])
                  )
                 )
                ),
             Seq (
              IfThenElse (
               Gt (Var "a", Int 0),
               Assign ("g", Var "a"),
               Assign ("g", Var "b")
              ),
              Assign ("n", Var "c")
             )
            )
           )
          );
   (*Procedure main *)
   Procedure ("main", [("a", IntSort); ("b", IntSort); ("c",
IntSort); ("d", IntSort)], [],
           ProcCall ("gcd", [
             Mult (Var "a", Var "b");
             Add (Var "a", Var "b");
```

```
                 Var "c";
                 Var "d"
               ])
            )
    ],
    "main",
    [("a", IntSort); ("b", IntSort); ("c", IntSort); ("d", IntSort)],
    ProcCall ("main", [Var "a"; Var "b"; Var "c"; Var "d"])
  )

let () =
  let input_vals = [IntVal 6; IntVal 4; IntVal 0; IntVal 0] in
  let result = run_program gcd_program input_vals in
  match result with
  | [IntVal g; IntVal n; _; _] ->
      Printf.printf "GCD = %d, Div Calls = %d\n" g n
  | _ -> Printf.printf "Error. \n"
```