A4 Ex5

```
(* TODO: Write more tests for the subst function. *)
let subst_tests : ((Exp.t * Identifier.t * Exp.t) * Exp.t) list =
  [
    ((parse "3", "x", parse "x + x"), parse "3 + 3");
    ((parse "1", "x", EVar "x"), parse "1");
    ((parse "1", "x", EVar "y"), EVar "y");
    ((parse "1", "x", parse "-x"), parse "-1");
    ((parse "1", "x", parse "x - x"), parse "1 - 1");
    ( (parse "True", "b", parse "if b then x else y"),
      parse "if True then x else y" );
    ((parse "1", "x", parse "if b then x else y"), parse "if b then 1 else y");
    ((parse "1", "y", parse "if b then x else y"), parse "if b then x else 1");
    ((parse "1", "x", parse "fun x -> x"), parse "fun x -> x");
    ((parse "1", "x", parse "fun y -> x"), parse "fun y -> 1");
    ((parse "1", "x", parse "let x be 2 in x"), parse "let x be 2 in x");
    ((parse "1", "x", parse "let y be 2 in x"), parse "let y be 2 in 1");
    ( (parse "2", "x", parse "let x:Num be x + x in x * x"),
      parse "let x:Num be 2 + 2 in x * x" );
    (* TODO: Write more tests for the subst function. *)
    (* Test substitutions for expressions with pairs *)
    ((parse "3", "x", parse "(x, x)"), parse "(3, 3)");
    (* Test substitutions for trival values *)
    ((parse "3", "x", parse "()"), parse "()");
    (* Test substitutions for expressions with let-pairs *)
    (* Hint: consider multiple cases including shadowing *)
    ((parse "1", "x", parse "let (x, y) be (x+3, z+5) in x+y"),
     parse "let (x, y) be (1+3, z+5) in x+y");
    ((parse "1", "z", parse "let (x, y) be (z+3, z+5) in x+y+z"),
     parse "let (x, y) be (1+3, 1+5) in x+y+1");
    (* Test substitutions for expressions with projections *)
    ((parse "True", "y", parse "(x, y).0"), parse "(x, True).0");
    ((parse "3", "x", parse "(x, y).1"), parse "(3, y).1");
    (* Test substitutions for expressions with injections *)
    ((parse "3", "x", parse "L (x)"), parse "L (3)");
    ((parse "False", "b", parse "R (b)"), parse "R (False)");
    (* Test substitutions for expressions in sum type *)
    ((parse "1", "x",
      parse "case L(x+2) of L(y)->y+x else R(b) -> if b then x else y"),
     parse "case L(1+2) of L(y)->y+1 else R(b) -> if b then 1 else y");
    ((parse "1", "x",
      parse "case L(x+2) of L(x)->y+x else R(b) -> if b then x else y"),
     parse "case L(1+2) of L(x)->y+x else R(b) -> if b then 1 else y");
    ((parse "1", "x",
```

```
     parse "case L(x+2) of L(y)->y+x else R(x) -> if b then x else y"),
    parse "case L(1+2) of L(y)->y+1 else R(x) -> if b then x else y");
   ((parse "1", "x",
     parse "case L(x+2) of L(x)->y+x else R(x) -> if b then x else y"),
    parse "case L(1+2) of L(x)->y+x else R(x) -> if b then x else y");

 ]

let eval_tests =
 [
   (parse "1", parse_val "1");
   (parse "-(1)", VNumLit (-1));
   (parse "1 + -1", parse_val "0");
   (parse "-1 + 1", parse_val "0");
   (parse "True", parse_val "True");
   (parse "if True then -1 else 0", VNumLit (-1));
   (parse "if False then -1 else 0", VNumLit 0);
   (parse "if 1 < 2 then 1 else 0", VNumLit 1);
   (parse "let x be 1 in x", parse_val "1");
   (parse "let x be 1 in -x", VNumLit (-1));
   (parse "(let incr:Num->Num be fun (x:Num) -> x + 1 in incr) 1", parse_val "2");
   (* TODO: Write more tests for the eval function. *)
   (* Test evaluations of expressions with pairs *)
   (parse "(2+3, 4*5)", VPair (parse_val "5", parse_val "20"));
   (* Test evaluations of trival values *)
   (parse "()", VTriv);
   (* Test evaluations of let-pair expressions *)
   (parse "let (x, y) be (3, 5) in x+y", parse_val "8");
   (* Test evaluations of expressions with projections *)
   (parse "(3, True).0", parse_val "3");
   (parse "(3, True).1", parse_val "True");
   (* Test evaluations of expressions with injections *)
   (parse "L (3)", VInjL (parse_val "3"));
   (parse "R (())", VInjR (VTriv));
   (* Test evaluations of function application *)
   (parse "fun (x:Num) -> x+2", VFun ("x", Some Num, parse "x+2"));
   (* Test evaluations of expressions in sum type *)
   (parse "case L(1+2) of L(x)->x+1 else R(b) -> if b then 9 else 7", parse_val
"4");
   (parse "case R(True) of L(x)->x+1 else R(b) -> if b then 9 else 7", parse_val
"9");
 ]

(* TODO: Replace `raise NotImplemented` with your code. *)
let rec subst (v : Exp.t) (x : Identifier.t) (e : Exp.t) : Exp.t =
  match e with
```

```
  | EVar y -> if x = y then v else e
  | ENumLit _ | EBoolLit _ -> e
  | EUnOp (o, e1) -> EUnOp (o, subst v x e1)
  | EBinOp (e1, o, e2) -> EBinOp (subst v x e1, o, subst v x e2)
  | EIf (e_cond, e_then, e_else) -> EIf (subst v x e_cond, subst v x e_then, subst
v x e_else)
  | EFun (y, ty_in, e_body) ->
      let e_body = (if x = y then Fun.id else subst v x) e_body in
      EFun (y, ty_in, e_body)
  | ELet (y, ty_def, e_def, e_body) ->
      let e_body = (if x = y then Fun.id else subst v x) e_body in
      ELet (y, ty_def, subst v x e_def, e_body)
  (* TODO: Implement these cases: *)
  | EPair (e_l, e_r) -> EPair (subst v x e_l, subst v x e_r)
  | ETriv -> e
  | ELetPair (y, z, e_def, e_body) ->
      let e_body = (if x = y || x = z then Fun.id else subst v x) e_body in
      ELetPair (y, z, subst v x e_def, e_body)
  | EPrjL e1 -> EPrjL (subst v x e1)
  | EPrjR e1 -> EPrjR (subst v x e1)
  | EInjL e1 -> EInjL (subst v x e1)
  | EInjR e1 -> EInjR (subst v x e1)
  | ECase (e_scrut, x_l, e_l, x_r, e_r) ->
      let e_l = (if x = x_l then Fun.id else subst v x) e_l in
      let e_r = (if x = x_r then Fun.id else subst v x) e_r in
      ECase (subst v x e_scrut, x_l, e_l, x_r, e_r)

(* TODO: Replace `raise NotImplemented` with your code. *)
let rec eval (e : Exp.t) : Value.t =
  match e with
  | EVar x -> raise IllTyped
  | ENumLit n -> VNumLit n
  | EBoolLit b -> VBoolLit b
  | EFun (x, ty, e_body) -> VFun (x, ty, e_body)
  | EIf (e_cond, e_then, e_else) ->
      if eval_bool e_cond then eval e_then else eval e_else
  | ELet (x, _, e_def, e_body) ->
      let v_def = Value.to_expr (eval e_def) in
      eval (subst v_def x e_body)
  | EUnOp (OpNeg, e) -> VNumLit (-(eval_num e))
  | EBinOp (e_l, ((OpPlus | OpMinus | OpTimes) as op), e_r) ->
      let f_op =
        match op with
        | OpPlus -> (+)
        | OpMinus -> (-)
        | _times -> ( * )
```

```
    in
    VNumLit (f_op (eval_num e_l) (eval_num e_r))
  | EBinOp (e_l, ((OpLt | OpEq | OpGt) as op), e_r) ->
    let f_op =
      match op with
      | OpLt -> (<)
      | OpEq -> (=)
      | _gt -> (>)
    in
    VBoolLit (f_op (eval_num e_l) (eval_num e_r))
  | EBinOp (e_fun, OpAp, e_arg) ->
    let (x, _, e_body) = eval_fun e_fun in
    let v_arg = Value.to_expr (eval e_arg) in
    eval (subst v_arg x e_body)
  (* TODO: Implement these cases: *)
  | EPair (e_l, e_r) -> VPair (eval e_l, eval e_r)
  | ETriv -> VTriv
  | ELetPair (x, y, e_def, e_body) ->
    (match eval e_def with
     | VPair (e_x, e_y) ->
        let v_x = Value.to_expr (e_x) in
        let v_y = Value.to_expr (e_y) in
        eval (subst v_y y (subst v_x x e_body))
     | _ -> raise IllTyped
    )
  | EPrjL e1 ->
    (match eval e1 with
     | VPair (e_l, e_r) -> e_l
     | _ -> raise IllTyped
    )
  | EPrjR e1 ->
    (match eval e1 with
     | VPair (e_l, e_r) -> e_r
     | _ -> raise IllTyped
    )
  | EInjL e1 -> VInjL (eval e1)
  | EInjR e1 -> VInjR (eval e1)
  | ECase (e_scrut, x_l, e_l, x_r, e_r) ->
    (match e_scrut with
     |EInjL el -> let vl = eval el in eval (subst (Value.to_expr vl) x_l e_l)
     |EInjR er -> let vr = eval er in eval (subst (Value.to_expr vr) x_r e_r)
     | _ -> raise IllTyped)
(* when a num is expected *)
and eval_num : Exp.t -> int = fun e ->
  match eval e with
  | VNumLit n -> n
```

```
  | _ -> raise IllTyped
(* when a bool is expected *)
and eval_bool : Exp.t -> bool = fun e ->
  match eval e with
  | VBoolLit b -> b
  | _ -> raise IllTyped
(* when a function is expected *)
and eval_fun : Exp.t -> (Identifier.t * Typ.t option * Exp.t) = fun e ->
  match eval e with
  | VFun (x, ty, e_body) -> (x, ty, e_body)
  | _ -> raise IllTyped
```