

#### A4 Ex3

(\* TODO: Write tests for the syn function. \*)

```
let ctx_test : Ctx.t = [  
  ("x", Num);  
  ("y", Num);  
  ("z", Num);  
  ("b", Bool);  
  ("tyann", Arrow (Num, Num));  
  ("typbn", Prod (Bool, Num));  
  ("tysnb", Sum (Num, Bool));  
  ("u", Unit);  
]
```

```
let syn_tests: ((Ctx.t * Exp.t) * Typ.t option) list = [  
  (* Test type synthesis for values *)  
  ((Ctx.empty, parse "5"), Some(Num));  
  ((Ctx.empty, parse "True"), Some(Bool));  
  ((Ctx.empty, parse "(30, True).1"), Some(Bool));  
  ((ctx_test, parse "(z, b)"), Some(Prod (Num, Bool)));  
  ((Ctx.empty, parse "L (1)"), None);  
  ((Ctx.empty, parse "()"), Some(Unit));  
  ((ctx_test, parse "fun x -> x + 1"), None);  
  ((ctx_test, parse "fun (b:Num) -> b + 1"), Some(Arrow (Num, Num)));  
  ((ctx_test, parse "u"), Some(Unit));  
  ((ctx_test, parse "typbn"), Some(Prod (Bool, Num)));  
  (* Test type synthesis for expressions with binary operators *)  
  ((ctx_test, parse "x * y"), Some(Num));  
  ((ctx_test, parse "- b"), None);  
  ((ctx_test, parse "x =? y"), Some(Bool));  
  ((ctx_test, parse "x < y"), Some(Bool));  
  ((ctx_test, parse "x > y"), Some(Bool));  
  ((ctx_test, parse "x < b"), None);  
  (* Test type synthesis for if-expressions and let-expressions *)  
  ((ctx_test, parse "let p be (z, True) in let (x, y) be p in if y then x else x+1"),  
  Some(Num));  
  ((Ctx.empty, parse "let f be fun (x:Num) -> x+1 in f (1)"), Some(Num));  
  ((Ctx.empty, parse "let p be (30, True) in p.0"), Some(Num));  
  ((ctx_test, parse "if b then x else y"), Some(Num));  
  ((ctx_test, parse "if u then x else y"), None);  
  ((ctx_test, parse "let x be 1 in x + y"), Some(Num));  
  ((ctx_test, parse "let u:Bool be 1 in u + y"), None);  
  ((ctx_test, parse "case tysnb of L(x) -> x+1 else R(b) -> if b then 3 else 5"),  
  Some(Num));  
  ((ctx_test, parse "case y of L(x) -> x+1 else R(b) -> if b then 3 else 5"), None);  
]
```

```
(*
 *)
]
```

```
(* TODO: Write tests for the ana function *)
```

```
let ana_tests: ((Ctx.t * Exp.t * Typ.t) * bool) list = [
  ((Ctx.empty, parse "5", Num), true);
  ((ctx_test, parse "u", Unit), true);
  ((ctx_test, parse "u", Num), false);
  ((ctx_test, parse "(b, u)", Prod(Bool, Unit)), true);
  ((Ctx.empty, parse "fun x -> x", Arrow(Num, Num)), true);
  ((Ctx.empty, parse "fun x -> x", Arrow(Bool, Bool)), true);
  ((Ctx.empty, parse "fun x -> x + 1", Arrow(Bool, Bool)), false);
  ((Ctx.empty, parse "fun (x:Bool) -> x", Arrow(Num, Num)), false);
  ((Ctx.empty, parse "fun (x:Bool) -> x", Arrow(Bool, Bool)), true);
  ((ctx_test, parse "let u:Bool be 1 in u + y", Num), false);
  ((ctx_test, parse "let u:Num be 1 in u + y", Num), true);
  ((ctx_test, parse "let u be x in u + y", Num), true);
  ((ctx_test, parse "let u be zz in u + y", Num), false);
  ((ctx_test, parse "let p be (z, True) in let (x, y) be p in if y then x else x+1",
Num), true);
  ((ctx_test, parse " let (x, y) be z in if y then x else x+1", Num), false);
  ((ctx_test, parse "if b then x else y", Num), true);
  ((ctx_test, parse "if u then x else y", Num), false);
  ((Ctx.empty, parse "L (1)", Sum(Num, Prod(Bool, Unit))), true);
  ((ctx_test, parse "case tysnb of L(x) -> x+1 else R(b) -> if b then 3 else 5",
Num), true);
  ((ctx_test, parse "case tysnb of L(x) -> x>0 else R(b) -> if b then 3 else 5",
Num), false);
  ((ctx_test, parse "case tysnb of L(x) -> x+1 else R(b) -> b", Num), false);
  ((ctx_test, parse "case y of L(x) -> x+1 else R(b) -> if b then 3 else 5", Num),
false);
  ((ctx_test, parse "case tysnb of L(x) -> x+1 else R(b) -> b+1", Num), false);
  ((ctx_test, parse "case tysnb of L(x) -> (if x then 4 else 6) else R(b) -> if b
then 3 else 5", Num), false);
  (* Test more type analysis *)
```

```
(* HINT 1: Test type analysis for expressions that
CAN be analyzed against different types defined in Typ.t *)
```

```
(* HINT 2: Test type analysis for expressions that
CAN NOT be analyzed against different types defined in Typ.t *)
```

```
]
let extract (o: Typ.t option): Typ.t =
  match o with
  | Some i -> i
  | None -> Unit
```

```

(* TODO: Finish implementing the syn function. *)
let rec syn (ctx: Ctx.t) (e: Exp.t): Typ.t option =
  match e with
  | EVar x -> Ctx.lookup ctx x (*S-Var*)
  | ENumLit _ -> Some Num (*S-NumLiteral*)
  | EBoolLit _ -> Some Bool (*S-True, S-False*)
  | ETriv -> Some Unit (*S-Triv*)
  | EFun (x, None, e_body) -> None
  | EFun (x, Some ty_in, e_body) -> (*S-FunAnn*)
    if (syn (Ctx.extend ctx (x, ty_in)) e_body) = None then None else
    Some (Arrow (ty_in, extract(syn (Ctx.extend ctx (x, ty_in)) e_body)))
  | EUnOp (OpNeg, e') -> (*S-Neg*)
    if ana ctx e' (Num: Typ.t) then Some Num else None
  | EBinOp (e_l, (OpPlus | OpMinus | OpTimes), e_r) -> (*S-Plus, S-Minus, S-
Times*)
    if ana ctx e_l Num && ana ctx e_r Num then Some Num else None
  | EBinOp (e_l, (OpLt | OpGt | OpEq), e_r) -> (*S-Lt, S-Gt, S-Eq*)
    if ana ctx e_l Num && ana ctx e_r Num then Some Bool else None
  | EBinOp (e_fun, OpAp, e_arg) -> (*S-Ap*)
    (match extract(syn ctx e_fun) with
    | Arrow (ty_in, ty_out) ->
      if ana ctx e_arg ty_in then Some ty_out else None
    | _ -> None)
  | EPair (e_l, e_r) -> (*S-Pair*)
    if (syn ctx e_l) <> None && (syn ctx e_r) <> None then
      Some (Prod (extract(syn ctx e_l), extract(syn ctx e_r)))
    else None
  | EPrjL e -> (*S-PrjL*)
    (match extract(syn ctx e) with
    | Prod (ty_l, ty_r) -> Some ty_l
    | _ -> None)
  | EPrjR e -> (*S-PrjR*)
    (match extract(syn ctx e) with
    | Prod (ty_l, ty_r) -> Some ty_r
    | _ -> None)
  | EInjL e -> None
  | EInjR e -> None
  | ECase (e_scrut, x_l, e_l, x_r, e_r) -> (*S-Case*)
    (match extract(syn ctx e_scrut) with
    | Sum (ty_l, ty_r) ->
      if (syn (Ctx.extend ctx (x_l, ty_l)) e_l) = (syn (Ctx.extend ctx (x_r, ty_r))
e_r)
      then syn (Ctx.extend ctx (x_l, ty_l)) e_l else None
    | _ -> None)
  | Elf (e_cond, e_then, e_else) -> (*S-If*)

```

```

    if ana ctx e_cond Bool && (syn ctx e_then) = (syn ctx e_else)
    then syn ctx e_then else None
| ELet (x, None, e_def, e_body) -> (*S-Let*)
    if (syn ctx e_def) <> None
    then syn (Ctx.extend ctx (x, extract(syn ctx e_def))) e_body else None
| ELet (x, Some ty_def, e_def, e_body) -> (*S-LetAnn*)
    if ana ctx e_def ty_def
    then syn (Ctx.extend ctx (x, ty_def)) e_body else None
| ELetPair (x, y, e_def, e_body) -> (*S-LetPair*)
    (match extract(syn ctx e_def) with
    | Prod (ty_l, ty_r) ->
        syn (Ctx.extend (Ctx.extend ctx (x, ty_l)) (y, ty_r)) e_body
    | _ -> None)
(* TODO: Finish implementing the ana function *)
and ana (ctx: Ctx.t) (e: Exp.t) (ty: Typ.t): bool =
match e with
| EFun (x, None, e_body) -> (*A-Fun*)
    (match ty with
    | Arrow (ty_in, ty_out) -> ana (Ctx.extend ctx (x, ty_in)) e_body ty_out
    | _ -> false
    )
| EFun (x, Some ty_in, e_body) -> (*A-FunAnn*)
    (match ty with
    | Arrow (ty_in', ty_out) -> if ty_in' = ty_in
        then ana (Ctx.extend ctx (x, ty_in)) e_body ty_out else false
    | _ -> false
    )
| EPair (e_l, e_r) -> (*A-Pair*)
    (match ty with
    | Prod (ty_l, ty_r) -> ana ctx e_l ty_l && ana ctx e_r ty_r
    | _ -> false
    )
| EInjL e -> (*A-InjL*)
    (match ty with
    | Sum (ty_l, ty_r) -> ana ctx e ty_l
    | _ -> false
    )
| EInjR e -> (*A-InjR*)
    (match ty with
    | Sum (ty_l, ty_r) -> ana ctx e ty_r
    | _ -> false
    )
| ECase (e_scrut, x_l, e_l, x_r, e_r) -> (*A-Case*)
    (match extract(syn ctx e_scrut) with
    | Sum (ty_l, ty_r) -> ana (Ctx.extend ctx (x_l, ty_l)) e_l ty &&
        ana (Ctx.extend ctx (x_r, ty_r)) e_r ty

```

```

| _ -> false)
| Elf (e_cond, e_then, e_else) -> ana ctx e_cond Bool && (*A-If*)
    ana ctx e_then ty &&
    ana ctx e_else ty
| ELet (x, None, e_def, e_body) -> (*A-Let*)
    if (syn ctx e_def) <> None
    then ana (Ctx.extend ctx (x, extract(syn ctx e_def))) e_body ty else false
| ELet (x, Some ty_def, e_def, e_body) -> (*A-LetAnn*)
    ana ctx e_def ty_def && ana (Ctx.extend ctx (x, ty_def)) e_body ty
| ELetPair (x, y, e_def, e_body) -> (*A-LetPair*)
    (match extract(syn ctx e_def) with
    | Prod (ty_l, ty_r) ->
        ana (Ctx.extend (Ctx.extend ctx (x, ty_l)) (y, ty_r)) e_body ty
    | _ -> false)
| _ -> (syn ctx e) = (Some ty) (*A-subsumption*)

```