# Ozyegin University
# CS 321 Programming Languages
# Sample Problems on Interpretation

1. (From PLC, Exercise 1.1) Given the definition of the simple ArithLang below, extend this language with conditional expressions (i.e. "if") corresponding to Java's expression $e_1$ `?` $e_2$ `:` $e_3$, or OCaml's `if` $e_1$ `then` $e_2$ `else` $e_3$. Evaluation of a conditional expression should evaluate $e_1$ first. If it yields a non-zero value, evaluate $e_2$, otherwise evaluate $e_3$.

```
type exp = CstI of int
         | Var of string
         | Add of exp * exp
         | Mult of exp * exp
         | Subt of exp * exp
         | Div of exp * exp
         | LetIn of string * exp * exp

(* lookup: string -> (string * int) list -> int *)
let rec lookup x env =
  match env with
  | [] -> failwith ("Unbound name " ^ x)
  | (y,i)::rest -> if x = y then i
                   else lookup x rest

(* eval: exp -> (string * int) list -> int *)
let rec eval e env =
  match e with
  | CstI i -> i
  | Var x -> lookup x env
  | Add(e1, e2)  -> eval e1 env + eval e2 env
  | Mult(e1, e2) -> eval e1 env * eval e2 env
  | Subt(e1, e2) -> eval e1 env - eval e2 env
  | Div(e1, e2)  -> eval e1 env / eval e2 env
  | LetIn(x, e1, e2) -> let v = eval e1 env
                        in let env' = (x, v)::env
                           in eval e2 env'
```

**Solution:** Here is the diff:

```
diff --git a/sampleProblems/interpretation/arith.ml b/sampleProblems/interpretation/arith.ml
index 17db0c0..bbc4556 100644
--- a/sampleProblems/interpretation/arith.ml
+++ b/sampleProblems/interpretation/arith.ml
@@ -5,6 +5,7 @@ type exp = CstI of int
           | Subt of exp * exp
           | Div of exp * exp
           | LetIn of string * exp * exp
+          | If of exp * exp * exp

 (* lookup: string -> (string * int) list -> int *)
 let rec lookup x env =
@@ -25,3 +26,6 @@ let rec eval e env =
    | LetIn(x, e1, e2) -> let v = eval e1 env
                          in let env' = (x, v)::env
                             in eval e2 env'
+  | If(e1, e2, e3) -> if (eval e1 env) = 0
+                      then eval e3 env
+                      else eval e2 env
```

2. (From PLC, Exercise 1.1) Extend ArithLang to handle three additional operators: "max", "min", and "=". Like the existing binary operators, they take two argument expressions. The equals operator should return 1 when true and 0 when false.

**Solution:** Here is the diff:

```
diff --git a/sampleProblems/interpretation/arith.ml b/sampleProblems/interpretation/arith.ml
index 17db0c0..921d4de 100644
--- a/sampleProblems/interpretation/arith.ml
+++ b/sampleProblems/interpretation/arith.ml
@@ -4,6 +4,9 @@ type exp = CstI of int
          | Mult of exp * exp
          | Subt of exp * exp
          | Div of exp * exp
+         | Min of exp * exp
+         | Max of exp * exp
+         | Eq of exp * exp
          | LetIn of string * exp * exp

 (* lookup: string -> (string * int) list -> int *)
@@ -22,6 +25,15 @@ let rec eval e env =
   | Mult(e1, e2) -> eval e1 env * eval e2 env
   | Subt(e1, e2) -> eval e1 env - eval e2 env
   | Div(e1, e2)  -> eval e1 env / eval e2 env
+  | Min(e1, e2)  -> let v1 = eval e1 env
+                    in let v2 = eval e2 env
+                       in if v1 < v2 then v1 else v2
+  | Max(e1, e2)  -> let v1 = eval e1 env
+                    in let v2 = eval e2 env
+                       in if v1 > v2 then v1 else v2
+  | Eq(e1, e2)   -> let v1 = eval e1 env
+                    in let v2 = eval e2 env
+                       in if v1 = v2 then 1 else 0
   | LetIn(x, e1, e2) -> let v = eval e1 env
                         in let env' = (x, v)::env
                            in eval e2 env'
```

3. Write the representation of the following ArithLang expressions using the `exp` data type.

   (a) `v * 5 - k + 6`

   **Solution:** `Add(Subt(Mult(Var "v", CstI 5), Var "k"), CstI 6)`

   (b) `x + y + z + p`

   **Solution:** `Add(Add(Add(Var "x", Var "y"), Var "z"), Var "p")`

   (c) `5 - (y - 3) * (g + 1)`

   **Solution:** `Subt(CstI 5, Mult(Subt(Var "y", CstI 3), Add(Var "g", CstI 1)))`

(d)
```
let x =
  let a = 5
  in let b = 8
     in a + b
in x * (let y = x + 2 in y)
```

**Solution:**

```
LetIn("x",
      LetIn("a", CstI 5,
            LetIn("b", CstI 8,
                  Add(Var "a", Var "b"))),
      Mult(Var "x",
           LetIn("y", Add(Var "x", CstI 2),
                 Var "y")))
```

4. Write an OCaml function named `simplify` that takes an `exp` and returns its simplified form based on the rules below:

| |
|---|
| $0 + e \to e$ |
| $e + 0 \to e$ |
| $e - 0 \to e$ |
| $1 \times e \to e$ |
| $e \times 1 \to e$ |
| $0 \times e \to 0$ |
| $e \times 0 \to 0$ |
| $e - e \to 0$ |

Remark: This problem is harder than it seems, because simplification of expressions may enable other simplifications, and I want to you to handle those cases, too. See the test cases.

```
# simplify (Mult(CstI 1,
                 Mult(Add(Add(CstI 1,
                              Subt(Var "x", Var "x")),
                          Add(CstI 4, CstI 6)),
                      CstI 1)));;
- : exp = Add(CstI 1, Add(CstI 4, CstI 6))

# simplify (Subt(CstI 0, Mult(Add(Var "x", CstI 0), CstI 0)));;
- : exp = CstI 0

# simplify (LetIn("a", CstI 4,
                  Subt(CstI 0,
                       Mult(Add(Var "x", CstI 0),
                            CstI 0))));;
- : exp = LetIn("a", CstI 4, CstI 0)

# simplify (Subt(Add(CstI 7, CstI 0),
                 Mult(Add(Var "x", CstI 0), CstI 0)));;
- : exp = CstI 7

# simplify (Div(Subt(CstI 0,
                     Mult(Add(Var "x", CstI 0), CstI 0)),
```

```
                  CstI 7));;
- : exp = Div(CstI 0, CstI 7)
```

**Solution:**

```
let rec simplify e =
  match e with
  | CstI i -> e
  | Var x -> e
  | Add(e1, e2) ->
     (match (simplify e1, simplify e2) with
      | CstI 0, e2' -> e2'
      | e1', CstI 0 -> e1'
      | e1', e2'    -> Add(e1', e2'))
  | Subt(e1, e2) ->
     (match (simplify e1, simplify e2) with
      | e1', CstI 0 -> e1'
      | e1', e2'    -> if e1' = e2' then CstI 0
                       else Subt(e1', e2'))
  | Mult(e1, e2) ->
     (match (simplify e1, simplify e2) with
      | CstI 1, e2' -> e2'
      | e1', CstI 1 -> e1'
      | CstI 0, e2' -> CstI 0
      | e1', CstI 0 -> CstI 0
      | e1', e2'    -> Mult(e1', e2'))
  | Div(e1, e2) -> Div(simplify e1, simplify e2)
  | LetIn(x, e1, e2) -> LetIn(x, simplify e1, simplify e2)
```