

# 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  $e_3$ .

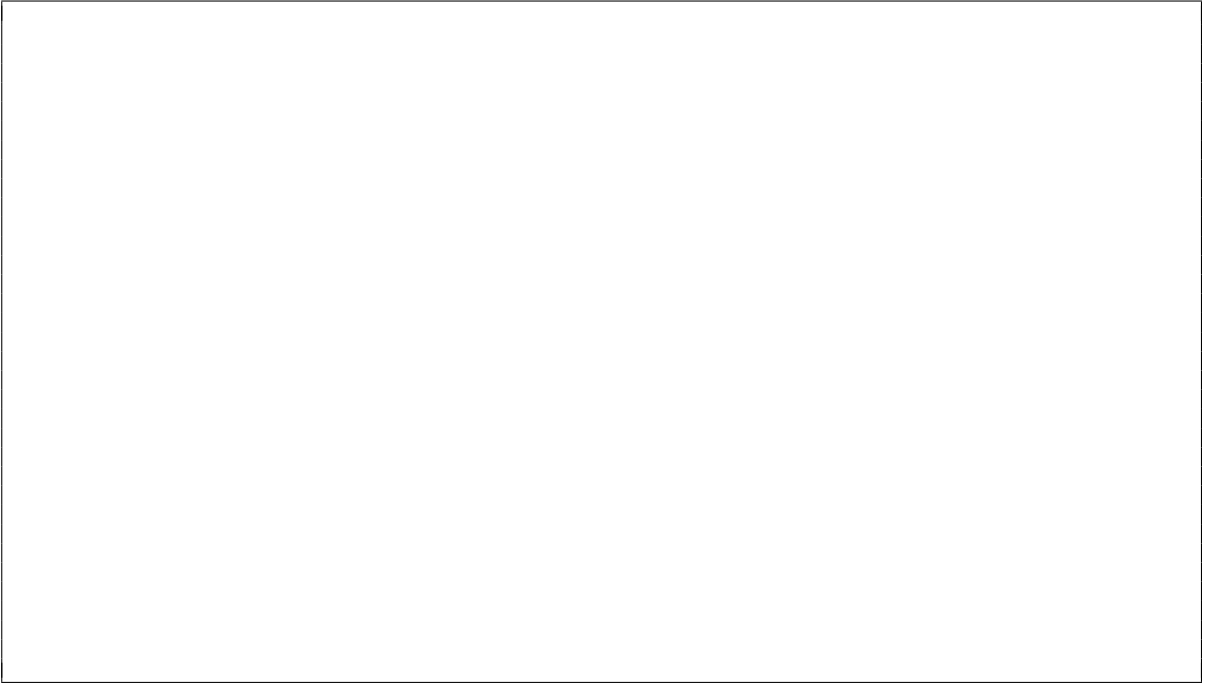
```

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'

```



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.

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

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

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

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

(d) 

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

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

$$\begin{array}{c} \hline 0 + e \rightarrow e \\ e + 0 \rightarrow e \\ e - 0 \rightarrow e \\ 1 \times e \rightarrow e \\ e \times 1 \rightarrow e \\ 0 \times e \rightarrow 0 \\ e \times 0 \rightarrow 0 \\ e - e \rightarrow 0 \\ \hline \end{array}$$

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)
```

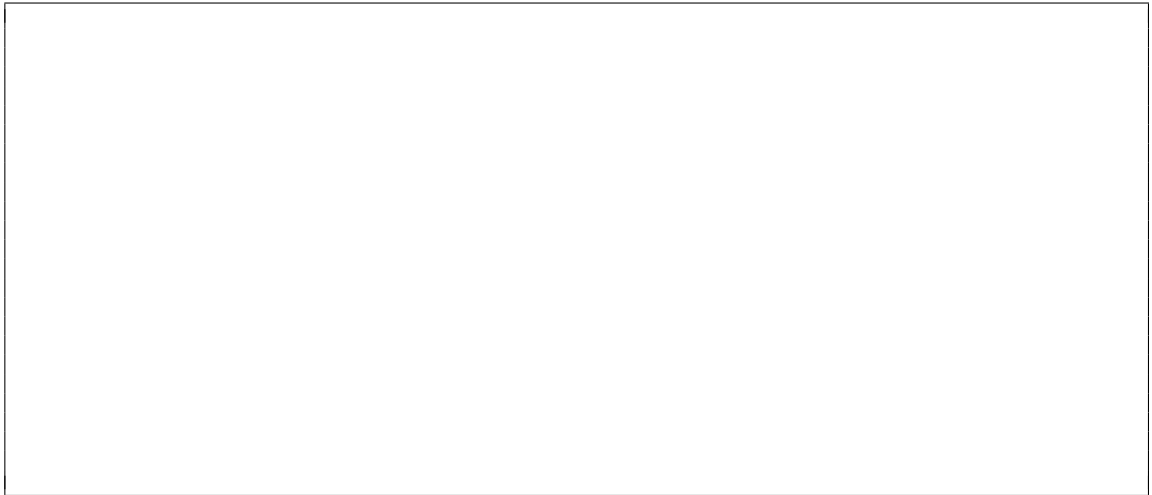
5. Is the grammar shown below ambiguous? If yes, give me an input that at least two different parse trees, and show those trees. If no, prove it.

```
main ::= exp EOF  
exp  ::= INT  
      | NAME  
      | exp PLUS exp  
      | exp STAR exp  
      | LET NAME EQ exp IN exp  
      | IF exp THEN exp ELSE exp
```

- 6.
- ```
main ::= exp EOF
exp  ::= INT
      | NAME
      | exp SLASH exp
      | exp PLUS exp
      | LET NAME EQ exp IN exp
      | IF exp THEN exp ELSE exp
```

Based on the grammar given above, show two different parse trees for the following inputs. For each, also state whether the ambiguity is related to **precedence** or **associativity**.

- (a)  $9 + 5 + 2$



(b)  $9 + 5 / 2$

