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:

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

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)
```

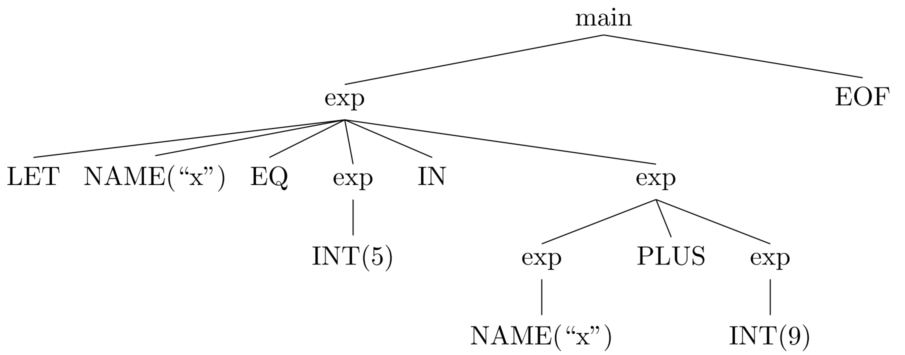
Solution:

```
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)
```

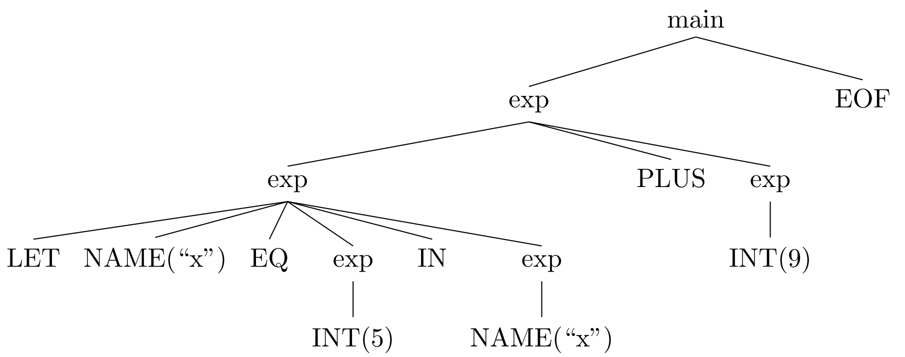
5. Is the grammar shown below ambiguous? If yes, give me an input that at least two different parse trees, and show those trees. If no, prove it.

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

Solution: It is ambiguous. Here is an input: `let x = 5 in x + 9`.



And the second tree is

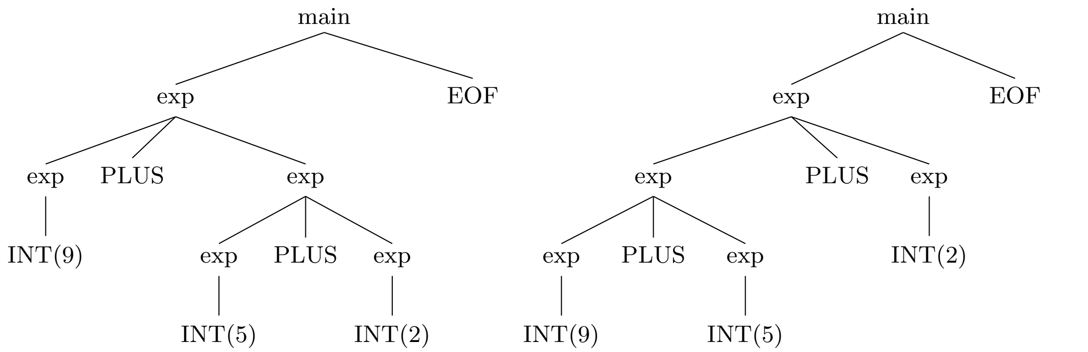


6.
```
main ::= exp EOF
exp  ::= INT
       | NAME
       | exp SLASH exp
       | exp PLUS exp
       | LET NAME EQ exp IN exp
       | IF exp THEN exp ELSE exp
```

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**.

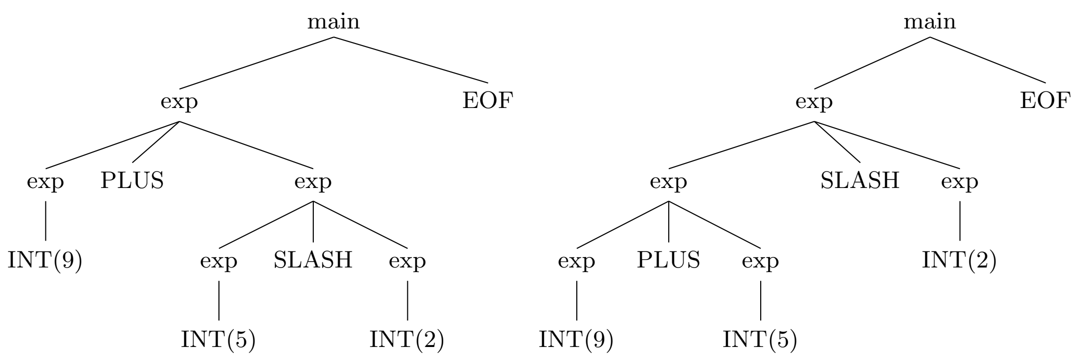
(a) `9 + 5 + 2`

Solution: This is related to associativity. Does the "+" sign associate to the left or to the right? That's the problem. If "+" associates to the right, we would get the tree on the left; if "+" associates to the left, we would get the tree on the right.



(b) `9 + 5 / 2`

Solution: This is related to precedence. Which operator has higher precedence, "/" or "+"? That is, who wins the fight over the ownership of "5"? That's the problem. If "/" has higher precedence, we would get the tree on the left; if "+" has higher precedence, we would get the tree on the right.

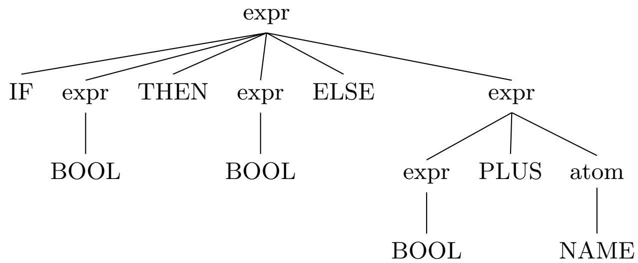


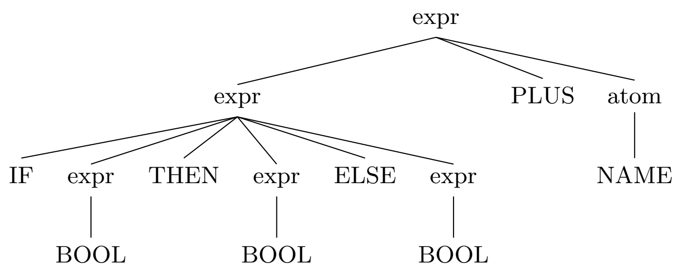
7. The following is an ambiguous grammar. Non-terminals in the notation are written using lowercase letters; terminals are all in capital letters. Give a term that has at least two different parse trees in this grammar. Show those two trees.

```
expr ::= expr PLUS atom
       | IF expr THEN expr ELSE expr
       | BOOL

atom ::= NAME
```

Solution: Consider "if true then true else true + x".





8. Write a lexer that recognizes all character sequences consisting of a and b where two a's are always separated by at least one b. For instance, these four strings are legal: b, a, ba, $ababbbaba$; but these two strings are illegal: aa, $babaa$. Your lexer should take a list of chars, and return true if the input is legal, otherwise return false.

Solution:

```
let rec lexer chars =
  match chars with
  | [] -> true
  | 'a'::'a'::rest -> false
  | _::rest -> lexer rest
```

9. Extend the Deve language interpreter to handle parenthesized expressions such as `(3 + 4) * 5`.

Solution: We need to first modify the lexer to recognize parentheses. For this, we will also add two more tokens:

```
type token = INT of int
           | ...
           | LPAR | RPAR

let rec tokenize chars =
  match chars with
  | ...
  | '('::rest -> LPAR::(tokenize rest)
  | ')'::rest -> RPAR::(tokenize rest)
  | ...
```

Now we have to update the parser. Because a parenthesized expression is "closed" on both sides with tokens, no ambiguity issues arise. For the same reason, a parenthesized expression is just like an "atomic" expression; it should be located at the highest level of precedence. So, we have:

```
and parseLevel4Exp tokens =
  match tokens with
  | ...
  | LPAR::rest ->
     let (e, tokens1) = parseExp rest in
     let rest2 = consume RPAR tokens1 in
     (e, rest2)
```

A parenthesized expression is just for grouping an expression; we simply return the `exp` we parse between parentheses. No new AST constructor is created.

10. Instead of having a separate AST constructor for each binary operator (e.g. `Add`, `Subt`, etc.), use a single constructor named `Binary` to handle any binary operator. For this, change the definition of the `exp` data type. In a `Binary`, in addition to the left and the right operands, keep the operator as a string.

    E.g. `Add(`e_1`, `e_2`)` becomes `Binary("+", `e_1`, `e_2`)`;
    `Mult(`e_1`, `e_2`)` becomes `Binary("*", `e_1`, `e_2`)`;
    `Subt(`e_1`, `e_2`)` becomes `Binary("-", `e_1`, `e_2`)`.

Solution: Here is the new definition for `exp`:

```
      type exp = CstI of int
         | CstB of bool
         | Var of string
         | Binary of string * exp * exp
         | LetIn of string * exp * exp
         | If of exp * exp * exp
```

Update the `eval` function as follows:

```
    let rec eval e env =
      match e with
      | ...
      | Binary(op, e1, e2)  ->
         let i1 = eval e1 env in
         let i2 = eval e2 env in
         (match op with
          | "+" -> i1 + i2
          | "-" -> i1 - i2
          | "*" -> i1 * i2
          | "/" -> i1 / i2
          | ...
         )
```

You will also need to modify the parser to return AST's according to this new definition:

```
...
and parseLevel2Exp tokens =
  let rec helper tokens e1 =
    match tokens with
    | PLUS::tok::rest ->
       (match tok with
        | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
                 in (Binary("+", e1, e2), tokens2)        (* CHANGED *)
        | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                 in (Binary("+", e1, e2), tokens2)        (* CHANGED *)
        | t   -> let (e2, tokens2) = parseLevel3Exp (tok::rest)
                 in helper tokens2 (Binary("+", e1, e2))  (* CHANGED *)
       )
    | MINUS::tok::rest ->
       (match tok with
        | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
                 in (Binary("-", e1, e2), tokens2)        (* CHANGED *)
        | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                 in (Binary("-", e1, e2), tokens2)        (* CHANGED *)
        | t   -> let (e2, tokens2) = parseLevel3Exp (tok::rest)
                 in helper tokens2 (Binary("-", e1, e2))  (* CHANGED *)
       )
    | _ -> (e1, tokens)
  in let (e1, tokens1) = parseLevel3Exp tokens in
     helper tokens1 e1

and parseLevel3Exp tokens =
  let rec helper tokens e1 =
    match tokens with
    | STAR::tok::rest ->
       (match tok with
        | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
                 in (Binary("*", e1, e2), tokens2)       (* CHANGED *)
        | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                 in (Binary("*", e1, e2), tokens2)       (* CHANGED *)
        | t   -> let (e2, tokens2) = parseLevel4Exp (tok::rest)
                 in helper tokens2 (Binary("*", e1, e2)) (* CHANGED *)
       )
    | SLASH::tok::rest ->
       (match tok with
        | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
                 in (Binary("/", e1, e2), tokens2)       (* CHANGED *)
        | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                 in (Binary("/", e1, e2), tokens2)       (* CHANGED *)
        | t   -> let (e2, tokens2) = parseLevel4Exp (tok::rest)
                 in helper tokens2 (Binary("/", e1, e2)) (* CHANGED *)
       )
    | _ -> (e1, tokens)
  in let (e1, tokens1) = parseLevel4Exp tokens in
     helper tokens1 e1
...
```

11. Extend the Deve interpreter (i.e. lexer, parser, and the eval function) to handle two relational operators: less-than (`<`) and less-than-or-equals (`<=`).

Solution: We need to first modify the lexer to recognize these new operators. This requires adding two new tokens as well.

```
type token = INT of int
           | ...
           | LESS | LESSEQ

let rec tokenize chars =
  match chars with
  | ...
  | '<'::'='::rest -> LESSEQ::(tokenize rest)
  | '<'::rest -> LESS::(tokenize rest)
  | ...
```

Note that I defined the `LESSEQ` case **above** the `LESS` case; otherwise it would not match.

Now we have to update the parser. Because the relational operators we're supposed to handle are just another binary operator (much like +, -, *, /), the ambiguity problems related to the existing binary operators apply to them as well. So, we need a specification of precedence and associativity:

- Relational operators are left-associative.
- Relational operators have lower precedence than addition and subtraction, but higher precedence than let-in and if-then-else.

So, our new table is:

Precedence	Rule	Operator	Associativity
1 (lowest)	let-in, if-then-else		-
1.5	relational	`<`,`<=`	left
2	plus, minus	`+`,`-`	left
3	star, slash	`*`,`/`	left
4 (highest)	atomic expressions		-

Essentially, we have created a new "level" of expressions. To avoid having to rename existing functions, let us call this level 1.5. So we add a new function named `parseLevel1_5Exp`. To write this, simply copy&paste `parseLevel2Exp`, and adapt as appropriate.

```
and parseLevel1Exp tokens =
  match tokens with
  | LET::rest -> parseLetIn tokens
  | IF::rest -> parseIfThenElse tokens
  | _ -> parseLevel1_5Exp tokens          (* CHANGED *)
...
and parseLevel1_5Exp tokens =             (* NEW FUNCTION *)
  let rec helper tokens e1 =
    match tokens with
    | LESS::tok::rest ->
       (match tok with
        | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
                 in (Binary("<", e1, e2), tokens2)
        | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                 in (Binary("<", e1, e2), tokens2)
        | t   -> let (e2, tokens2) = parseLevel2Exp (tok::rest)
```

```
                  in helper tokens2 (Binary("<", e1, e2))
        )
     | LESSEQ::tok::rest ->
        (match tok with
         | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
                  in (Binary("<=", e1, e2), tokens2)
         | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                  in (Binary("<=", e1, e2), tokens2)
         | t   -> let (e2, tokens2) = parseLevel2Exp (tok::rest)
                  in helper tokens2 (Binary("<=", e1, e2))
        )
     | _ -> (e1, tokens)
   in let (e1, tokens1) = parseLevel2Exp tokens in
      helper tokens1 e1
```

Finally, we extend the implementation of the `eval` function to handle these new operators as well.

```
let rec eval e env =
  match e with
  | ...
  | Binary(op, e1, e2)  ->
     let i1 = eval e1 env in
     let i2 = eval e2 env in
     (match op with
      | ...
      | "<" -> if i1 < i2 then 1 else 0      (* NEW CASE *)
      | "<=" -> if i1 <= i2 then 1 else 0    (* NEW CASE *)
     )
```

12. Change the definition of the interpreter so that boolean values are not handled as 0 and 1, but handled separately as `true` and `false`. You will need to define a new data type named, say, `value`, for this. The `eval` function should now return a `value`, instead of an `int`.

Solution:

```
(* exp does not change *)

type value = Int of int
           | Bool of bool

(* lookup is polymorphic, does not need to change *)

(* eval: exp -> (string * value) list -> value *)
let rec eval e env =
  match e with
  | CstI i -> Int i
  | CstB b -> Bool b
  | Var x -> lookup x env
  | Binary(op, e1, e2)  ->
     let v1 = eval e1 env in
     let v2 = eval e2 env in
     (match op, v1, v2 with
      | "+", Int i1, Int i2 -> Int(i1 + i2)
```

```
      | "-", Int i1, Int i2 -> Int(i1 - i2)
      | "*", Int i1, Int i2 -> Int(i1 * i2)
      | "/", Int i1, Int i2 -> Int(i1 / i2)
      | "<", Int i1, Int i2 -> Bool(i1 < i2)
      | "<=", Int i1, Int i2 -> Bool(i1 <= i2)
     )
  | LetIn(x, e1, e2) -> let v = eval e1 env
                        in let env' = (x, v)::env
                           in eval e2 env'
  | If(e1, e2, e3) -> (match eval e1 env with
                       | Bool true -> eval e2 env
                       | Bool false -> eval e3 env
                       | _ -> failwith "Condition should be a Bool.")
```

Let's test:

```
# run "3 < 4";;
- : value = Bool true
# run "if 7 <= 4 then 42 else 34 + 1";;
- : value = Int 35
```

13. Extend the language with pairs: (e_1, e_2) and the `fst`, `snd` functions: `fst`(e), `snd`(e)

    E.g: `let p = (6+8, 9-5) in fst(p) + snd(p)` should evaluate to `Int 18`.

    You will need to extend the definition of `value` for this.

    E.g: `let p = (6+8, 9-5) in (snd(p), fst(p))` should evaluate to `Pair(Int 4, Int 14)`.

    Another example: `let p = (6+8, 9-5) in (snd(p), (fst(p) < 10, 5))` evaluates to `Pair(Int 4, Pair(Bool false, Int 5))`

    You can treat `fst` and `snd` as unary operators (i.e. operators that take a single argument).

Solution: First, the lexer. We already recognize the parentheses. Good. We do not recognize the comma, though. Also recognize `fst` and `snd` as keywords:

```
type token = INT of int
           | ...
           | COMMA | FST | SND

let keyword s =
  match s with
  | ...
  | "fst" -> FST
  | "snd" -> SND
  | _ -> NAME s

let rec tokenize chars =
  match chars with
  | ...
  | ','::rest -> COMMA::(tokenize rest)
  | ...
```

That was simple. Let's modify the parser.

```
and parseLevel4Exp tokens =
  match tokens with
  | ...
  | LPAR::rest ->
     let (e1, tokens1) = parseExp rest in
     (match tokens1 with
      | RPAR::rest1 -> (e1, rest1)
      | COMMA::rest1 ->
         let (e2, tokens2) = parseExp rest1 in
         let rest2 = consume RPAR tokens2 in
         (Binary(",", e1, e2), rest2)
     )
  | FST::LPAR::rest ->
     let (e, tokens1) = parseExp rest in
     let rest1 = consume RPAR tokens1 in
     (Unary("fst", e), rest1)
  | SND::LPAR::rest ->
     let (e, tokens1) = parseExp rest in
     let rest1 = consume RPAR tokens1 in
     (Unary("snd", e), rest1)
  | ...
```

Finally, the evaluator:

```
type exp = CstI of int
         | ...
         | Unary of string * exp           (* NEWLY ADDED *)
         | Binary of string * exp * exp
         | ...

type value = Int of int
           | Bool of bool
           | Pair of value * value         (* NEWLY ADDED *)

let rec eval e env =
  match e with
  | ...
  | Unary(op, e) ->                        (* NEW CASE *)
     let v = eval e1 env in
     (match op, v with
      | "fst", Pair(v1, v2) -> v1
      | "snd", Pair(v1, v2) -> v2
     )
  | Binary(op, e1, e2)  ->
     let v1 = eval e1 env in
     let v2 = eval e2 env in
     (match op, v1, v2 with
      | ...
      | ",", _, _ -> Pair(v1, v2)           (* NEW CASE *)
     )
```

Let's test:

```
# run "let p = (6+8, 9-5) in fst(p) + snd(p)";;
- : value = Int 18
# run "let p = (6+8, 9-5) in (snd(p), fst(p))";;
- : value = Pair (Int 4, Int 14)
# run "let p = (6+8, 9-5) in (snd(p), (fst(p) < 10, 5))";;
- : value = Pair (Int 4, Pair (Bool false, Int 5))
```

14. Extend the language to handle a simple match expression for pairs in the following form:

    `match` e_1 `with` (x, y) `->` e_2

    Here, e_1 and e_2 are arbitrary expressions, x and y are arbitrary names. e_1 is expected to evaluate to a pair. x and y may be used inside e_2; so the match expression should bind x and y to the first and second item, respectively, of the pair that we obtain from evaluating e_1.

    E.g. `match (5+6, 2*3) with (f,s) -> f + s` evaluates to `Int 17`.

15. Extend the language with the boolean negation (i.e. logical-not) operator: `not(`e`)`. For simplicity of parsing, we require parentheses here. So, there are no ambiguity risks.

16. Extend the language with the greater-than-or-equal-to operator: e_1 `>=` e_2.

    Do NOT change the definition of the `eval` function for this. Instead, simply parse a `>=` as a logical-NOT of a `<`. E.g. e_1 `>=` e_2 should be parsed as if it were `not(`e_1 `<` e_2`)`. Note that our language already handles `<` and `not`.

17. Extend the language with two more binary operators, `min` and `max`, with the following syntax: `min(`e_1`,` e_2`)` and `max(`e_1`,` e_2`)`. You can still use the `Binary` constructor for `min` and `max` although they are not infix operators.