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:

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

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)
```

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`

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
```

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.

The questions below are based on Deve 1.0, given at https://github.com/aktemur/cs321/tree/master/Deve-1.0. The code is also shown below:

EVALUATOR:

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

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
  | CstB b -> if b then 1 else 0
  | 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'
  | If(e1, e2, e3) -> (match eval e1 env with
                      | 0 -> eval e3 env
                      | m -> eval e2 env)
```

LEXER:

```
(* This is the lexer.
   The goal of the lexer is to take a string
   and recognize the tokens in it.
   A "token" is a categorized unit
   of input, such as "an integer", "the plus operator",
   "a name", etc.
 *)
type token = INT of int
           | BOOL of bool
```

```
              | NAME of string
              | PLUS | STAR | MINUS | SLASH
              | LET | EQUALS | IN
              | IF | THEN | ELSE
              | ERROR of char
              | EOF
;;

let isDigit c = '0' <= c && c <= '9'

let digitToInt c = int_of_char c - int_of_char '0'

let isLowercaseLetter c = 'a' <= c && c <= 'z'

let isUppercaseLetter c = 'A' <= c && c <= 'Z'

let isLetter c = isLowercaseLetter c || isUppercaseLetter c

let charToString c = String.make 1 c

let keyword s =
  match s with
  | "let" -> LET
  | "in" -> IN
  | "if" -> IF
  | "then" -> THEN
  | "else" -> ELSE
  | "true" -> BOOL true
  | "false" -> BOOL false
  | _ -> NAME s

(*  tokenize: char list -> token list  *)
let rec tokenize chars =
  match chars with
  | [] -> [EOF]
  | '+'::rest -> PLUS::(tokenize rest)
  | '*'::rest -> STAR::(tokenize rest)
  | '-'::rest -> MINUS::(tokenize rest)
  | '/'::rest -> SLASH::(tokenize rest)
  | '='::rest -> EQUALS::(tokenize rest)
  | ' '::rest -> tokenize rest
  | '\t'::rest -> tokenize rest
  | '\n'::rest -> tokenize rest
  | c::rest when isDigit(c) ->
     tokenizeInt rest (digitToInt c)
  | c::rest when isLowercaseLetter(c) ->
     tokenizeName rest (charToString c)
  | c::rest -> (ERROR c)::(tokenize rest)

and tokenizeInt chars n =
  match chars with
  | c::rest when isDigit(c) ->
     tokenizeInt rest (n * 10 + (digitToInt c))
```

```
  | _ -> (INT n)::(tokenize chars)

and tokenizeName chars s =
  match chars with
  | c::rest when isLetter(c) || isDigit(c) ->
     tokenizeName rest (s ^ (charToString c))
  | _ -> (keyword s)::(tokenize chars)
;;

let chars_of_string s =
  let rec helper n acc =
    if n = String.length s
    then List.rev acc
    else let c = String.get s n
         in helper (n+1) (c::acc)
  in helper 0 []
;;

let scan s =
  tokenize (chars_of_string s)
;;
```

PARSER:

```
(* A helper function to convert a token to a string *)
let toString tok =
  match tok with
  | INT i -> "INT(" ^ string_of_int i ^ ")"
  | BOOL b -> "BOOL(" ^ string_of_bool b ^ ")"
  | NAME x -> "NAME(\"" ^ x ^ "\")"
  | PLUS -> "PLUS"
  | STAR -> "STAR"
  | MINUS -> "MINUS"
  | SLASH -> "SLASH"
  | LET -> "LET"
  | EQUALS -> "EQUALS"
  | IN -> "IN"
  | IF -> "IF"
  | THEN -> "THEN"
  | ELSE -> "ELSE"
  | ERROR c -> "ERROR('" ^ (charToString c) ^ "')"
  | EOF -> "EOF"

(* consume: token -> token list -> token list
   Enforces that the given token list's head is the given token;
   returns the tail.
*)
let consume tok tokens =
  match tokens with
  | [] -> failwith ("I was expecting to see a " ^ (toString tok))
```

```
  | t::rest when t = tok -> rest
  | t::rest -> failwith ("I was expecting a " ^ (toString tok) ^
                          ", but I found a " ^ toString(t))

(* parseExp: token list -> (exp, token list)
   Parses an exp out of the given token list,
   returns that exp together with the unconsumed tokens.
 *)
let rec parseExp tokens =
  parseLevel1Exp tokens

and parseLevel1Exp tokens =
  parseLETorIForOther parseLevel2Exp tokens

and parseLETorIForOther otherParseFun tokens =
  match tokens with
  | LET::rest -> let (e, tokens2) = parseLetIn tokens
                 in (e, tokens2)
  | IF::rest  -> let (e, tokens2) = parseIfThenElse tokens
                 in (e, tokens2)
  | _         -> let (e, tokens2) = otherParseFun tokens
                 in (e, tokens2)

and parseLetIn tokens =
  match tokens with
  | LET::NAME(x)::EQUALS::rest ->
     let (e1, tokens1) = parseExp rest in
     let tokens2 = consume IN tokens1 in
     let (e2, tokens3) = parseExp tokens2 in
     (LetIn(x, e1, e2), tokens3)
  | _ -> failwith "Should not be possible."

and parseIfThenElse tokens =
  let rest = consume IF tokens in
  let (e1, tokens1) = parseExp rest in
  let tokens2 = consume THEN tokens1 in
  let (e2, tokens3) = parseExp tokens2 in
  let tokens4 = consume ELSE tokens3 in
  let (e3, tokens5) = parseExp tokens4 in
  (If(e1, e2, e3), tokens5)

and parseLevel2Exp tokens =
  let rec helper tokens e1 =
    match tokens with
    | PLUS::rest ->
       let (e2, tokens2) = parseLETorIForOther parseLevel3Exp rest
       in helper tokens2 (Add(e1, e2))
    | MINUS::rest ->
       let (e2, tokens2) = parseLETorIForOther parseLevel3Exp rest
       in helper tokens2 (Subt(e1, e2))
    | _ -> (e1, tokens)
  in let (e1, tokens1) = parseLevel3Exp tokens in
     helper tokens1 e1
```

```
and parseLevel3Exp tokens =
  let rec helper tokens e1 =
    match tokens with
    | STAR::rest ->
       let (e2, tokens2) = parseLETorIForOther parseLevel4Exp rest
       in helper tokens2 (Mult(e1, e2))
    | SLASH::rest ->
       let (e2, tokens2) = parseLETorIForOther parseLevel4Exp rest
       in helper tokens2 (Div(e1, e2))
    | _ -> (e1, tokens)
  in let (e1, tokens1) = parseLevel4Exp tokens in
     helper tokens1 e1

and parseLevel4Exp tokens =
  match tokens with
  | INT i :: rest -> (CstI i, rest)
  | NAME x :: rest -> (Var x, rest)
  | BOOL b :: rest -> (CstB b, rest)
  | t::rest -> failwith ("Unsupported token: " ^ toString(t))
  | [] -> failwith "No more tokens???"

(* parseMain: token list -> exp *)
let parseMain tokens =
  let (e, tokens1) = parseExp tokens in
  let tokens2 = consume EOF tokens1 in
  if tokens2 = [] then e
  else failwith "Oops."

(* parse: string -> exp *)
let rec parse s =
  parseMain (scan s)
```

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

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`)`.

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 (`<=`).

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`.

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).

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`.

    Match expression is at the same level of precedence with let-in and if-then-else.

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`.

> At this point, we have reached Deve 2.0, also available at `https://github.com/aktemur/cs321/tree/master/Deve-2.0`.