```
// ------------------------------------------------------------
//        Homework 4
//        Adam Levy, Alejandro Palacios, & Alicia Rodriguez
//        November 14, 2016
// ------------------------------------------------------------

// 1. An interesting use of first-class functions and ref cells in F# is to
// create a monitored version of a function:
(*
  > let makeMonitoredFun f =
      let c = ref 0
      (fun x -> c := !c+1; printf "Called %d times.\n" !c; f x);;
  val makeMonitoredFun : ('a -> 'b) -> ('a -> 'b)
  > let msqrt = makeMonitoredFun sqrt;;
  val msqrt : (float -> float)
  > msqrt 16.0 + msqrt 25.0;;
  Called 1 times.
  Called 2 times.
  val it : float = 9.0
*)
// First, explain why F# does not allow the following declaration:
(*
  let mrev = makeMonitoredFun List.rev
*)
// Now suppose we rewrite the declaration using the technique of eta expansion:
(*
  let mrev = fun x -> (makeMonitoredFun List.rev) x
*)
// Does this solve the problem? Explain why or why not.

// Solution

// The following declaration:
(*
  let mrev = makeMonitoredFun List.rev
*)
// returns the following error:
(*
  Value restriction. The value 'mrev' has been inferred to have generic type
      val mrev : ('_a list -> '_a list)
  Either make the arguments to 'mrev' explicit or, if you do not intend for it
  to be generic, add a type annotation.
*)
```

```
// This is because List.rev is not given a specific type and it is not given
// another parameter to input a list.

// When rewriting the declaration using the technique of eta expansion:
(*
  let mrev = fun x -> (makeMonitoredFun List.rev) x
*)
// An error is not returned, which would solve that problem, as shown below:
(*
  val mrev : x:'a list -> 'a list
*)
// However, when using a similar pattern, as used in msqrt, to call the
// function, the following is returned:
(*
  mrev [3;2;1] @ mrev [1..3];;
  Called 1 times.
  Called 1 times.
  val it : int list = [1; 2; 3; 3; 2; 1]
*)
// Which is not correct. Clearly, the function is called twice, but by using
// the technique of eta expansion, the counter gets reset everytime the function
// is called. This also happens when rewriting the msqrt function using
// the technique of eta expansion:
(*
  > let msqrt = fun x -> (makeMonitoredFun sqrt) x;;

  val msqrt : x:float -> float

  > msqrt 16.0 + msqrt 25.0;;
  Called 1 times.
  Called 1 times.
  val it : float = 9.0
*)

// So, in order to satisfy F#’s value restriction in declarations such as
// “let f = e”, ‘e’ must be of a restricted form called a “syntactic value”.
// One such syntactic value is a function declaration of the form
// (fun x -> [x]) where the value of x can be obtained without calculation.
// The eta expansion above satisfies that criteria and therefore satisfies F#’s
// value restriction. Furthermore, mrev will have a polymorphic type
// “a list -> a list”.

// Therefore, using the technique of eta expansion solves the problem of F#
```

```
// allowing the function declaration and the value restriction error. However,
// the function does not return the correct result.
```

// 2. Recall the unambiguous grammar for arithmetic expressions discussed in
// class:

```
(*
  E -> E+T | E-T | T
  T -> T*F | T/F | F
  F -> i | (E)
*)
```

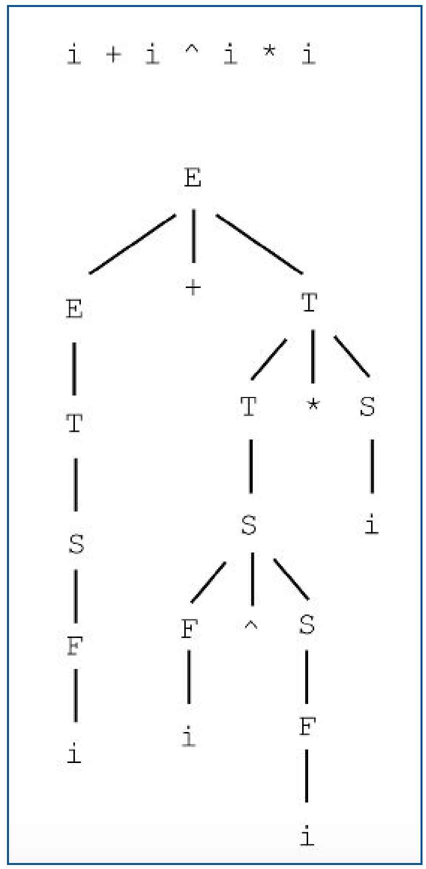
// Modify this grammar to allow an exponentiation operator, ^, so that we can
// write expressions like i+i^i*i. Of course, your modified grammar should be
// unambiguous. Give exponentiation higher precedence than the other binary
// operators and (unlike the other binary operators) make it associate to
// the right.

// Solution

// The new grammar gives higher precedence than the other
// binary operators by having it further down the context free
// grammar. The associativity to the right comes from having
// F evaluated first in the exponentiation portion of the
// grammar.

```
(*
  E -> E+T | E-T | T
  T -> T*S | T/S | S
  S -> F^S | F
  F -> i | (E)
*)
```

// Image of parse tree -->



// 3. Recall the grammar for the tiny language used in our discussion of
// recursive-descent parsing. Let us extend the grammar to allow both if-then
// and if-then-else statements:
(*
  S -> if E then S | if E then S else S | begin S L | print E
  L -> end | ; S L
  E -> i
*)
// Show that the grammar is now ambiguous.

// Solution

// The problem with this CFG arrives when you are attempting to nest an
// if-then-else statement within an if-then statement (or vice versa). This
// would cause the following ambiguity:
// 1.) if a < 4 then (if b > 3 then print 2) else print 3
// 2.) if a < 4 then (if b > 3 then print 2 else print 3)
// Both are correct according to this grammar, but if you are trying to nest an
// if-then-else statement within an if-then statement the second parse would be
// correct. If the opposite was the case, the first parse would be correct.

```
// 4. Following the approach described in class, write a complete (pseudo-code)
// recursive-descent parser for the grammar in question 3, including functions
// S(), L(), and E(). Try to improve on the code given in class by returning
// helpful error messages.
// [You may wonder how a recursive-descent parser can be written, given the
// ambiguity that you found in question 3. This ambiguity, known as the
// dangling else ambiguity, is found in many programming languages, including
// C and Java. However, you should see that the parser can resolve that
// ambiguity in a natural way that corresponds to a rule that you should
// have learned in your study of C or Java.]

// Solution

// Modified CFG:
// S -> if E then S | else S | begin S L | print E
// L -> end | ; S L
// E -> i

// **THIS CODE WAS COPIED FROM THE NOTES AND MODIFIED AS NEEDED**


 // lookahead token
 int tok = nextToken();

 // log error function to return error messages
 void logError(String s) {printf(s); error();}

 void advance() {tok = nextToken();}

 // used whenever a specific token t must appear next
 void eat(int t) {
   if (tok == t) advance();
   else logError("eat(): Token is not equal to input t");
 }

 void S() {
   switch (tok) {
     case IF: advance(); E(); eat(THEN); S(); break;
     case ELSE: advance(); S(); break;
     case BEGIN: advance(); S(); L(); break;
     case PRINT: advance(); E(); break;
     default: logError("S(): Case not found");
   }
```

```
}

void L() {
  switch (tok) {
    case END: advance(); break;
    case SEMICOLON: advance(); S(); L(); break;
    default: logError("L(): Case not found");
  }
}

void E() {
  eat(ID);
}

void main() {
  S();
  if (tok == EOF) accept();
  else logError("Token is not EOF");
}
```