Tree traversals and **recursions** are almost identical.

Recursion Tutorial:

```
g(x) {
        return x*2;
}

f(y){
        return g(y) - 1;    // we call function g here.

// When you write recursion function, you call the function you are implementing. While writing
it inside itself, you just assume that it works - even if you did not completely implement it.
For example, in above code, if you implement f() before implementing g() with the assumption
that g() multiplies by 2, it is okay.
}

fact(x){
        if x == 0
                return 1;
        else
                return x * fact(x - 1);    //we have recursion here.

// We assume this function fact(x - 1) works correctly. It is like how induction in mathematics
works. In induction, you assume some step k works. Then, you prove step k+1 works. Then, it
proves it works any value of k.
Invariant: fact(x - 1)'s result is (x-1)!. You just assume that invariant, while implementing fact()
itself.
For example, while implementing function block() last lecture, we just assumed that the
functions we called inside block() are implemented. After you implement the others,
everything will work.



}
```

You will compile recursive programs. However, machine does not have recursion. You need to translate recursion into something that simulates recursion.
Everytime you call the function, you call a new function.
Rather than thinking fact(3) is fact(x), think of it as a new function fact3()
So, if you call fact(3), think of it like you called fact3(), fact2(), fact1(), fact0(), respectively:

```
fact3():
  3 * fact2()

fact2():
  2 * fact1() ….
```

It is easier to trace the recursion by thinking this way. If we create two separate functions we do not have recursion anymore.
fact3() calls fact2()
fact2() calls fact1()
fact1() calls fact0().

You are simulating call stack. It is how recursion is simulated in a machine where recursion is not implemented.

```
                      mergesort()
                    /                \
          mergesort()            mergesort()
          /           \                  …..
mergesort()      mergesort()          …
…..
```

If you draw the call graph, you just get a tree structure.

AST: In AST you have fewer nodes. You abstract away the stuff you do not need while you are processing the language.

Parsing:

- you can do piece by piece implementation in your parser, too. First, assume the functions you are using are correctly implemented. Later, implement these functions.
- You can use print statements to follow your call trace. There are fancy ways to do that. However, this approach should be easier for you.
- Everytime we recognize one of the structures in AST structure, we create and add this node, so that, we create our parse tree.
- You are traversing the parse tree and constructing the AST nodes in parser.

Keep the invariant: pick your conditions and stick with them.

Return types for all of our functions are AST nodes. They are already given as return types.
You have the list of tokens which is being consumed. token list is a global variable. You change its state by using **next()** and **previous()** functions. Before you consume any token, you need to check it.
ensure_token() gives you a parser error if the current token does not match with the parameter (which is expected if the input PL0 program is not valid)
Precondition invariant: The pointer to the token list should be on just before the token that our production suggests. When you enter to a function, you have to first call **next()** to get to the next token to continue with.
var x : int
var result : int
For example, when you are doing parsing for the structure starting with “var result : int”, your token pointer will be pointing to “int” at the first line. Hence, you need call next() at first.

- Notice that we don’t call **next()** or **previous()** in block. The reason is that it does not consume any tokens. Hence, it is enough to call the non-terminals functions.

Postcondition invariant: The parser function should not consume any more tokens that it needs to consume.
The reason you call previous() is that you have a look at the token by calling next(). Then, if it is not what you are looking for, you call previous() to push the token back to the token list.

ExpressionList():

```
ExpressionList{
next();
if (is_token(RPAREN)){
        //do empty production
}else{
        expr();
        next(); // because the pointer is just before the COMMA token, and we need to read
                // COMMA
        while(is_token(COMMA)){
                exp();

                next();
        }
        previous()
}
return node;
}
```

- Note: sit down and step through the program one token at a time. You always have to be careful about where your pointer is.

Now, let's create AST node:

- Every function is supposed to return the AST node of its subtree.
  Hence, we need to use addExpression() to add the return value of expr() to ExpressionList node.
- It easier to do parsing first before AST.

```
ExpressionList{
next();
if (is_token(RPAREN)){      // no arguments
          previous()            //do empty production
}else{                          // has arguments

          previous()
          addExpression(node, expr());
          next();
          while(is_token(COMMA)){
                    addExpression(node, expr());

                    next();
          }
          previous()
}
return node;
}
```

You should be able to work with the given API to complete the project.
(Check grammar.md, exprlist has a fix)
exprlist ::= [expr { COMMA expr }]

How can we check if "**exprlist**" is really empty?

- It needs a careful analysis of the grammar to come up with the trick: check for RPAREN **"exprlist"** is used in two places. In both, it is always followed by RPAREN. Therefore, we check for RPAREN. If there is, we know that **"exprlist"** must be empty
- Track the parsing "**exprlist**" by using two different examples: f() and f(x)
  You will see that how RPAREN trick works and how we keep invariant by calling previous() at the end.

Expression():

```
static struct  Expression *expr() {
  // partially given to enable write NUMBER (allow user to work with tagged union)

  struct Expression *left = simpleexpr();

  next();
  if (is_relop()) {
    struct Expression *node = newExpression(BINARYEXPRESSION);

    node->binary_left = left;
    node->binary_op = token();
    node->binary_right = simpleexpr();
    return node;
  } else {
    previous();
    return left;
  }
}
```

- **is_relop():** checks if the current token is a relational operator
- For **expr()**, we first create the left child. If we do not have any right child (if it is not a binary expression), we directly return left since it is the expr() itself.
- **token()** gets you the token.
- Use **print_token()** to print out the tokens for debugging
  print_token(filepointer, token)
  For example, to write the current token, you can use: print_token(stderr, token())

Once you get the parser done, you will be very good at tree traversal.
LR parsing can create parse tree from any unambiguous context free grammar.