Parser+ParseTreeBuilder P2

P2

150 (75 for just parsing and 75 for building the tree). Submission command:
`/accounts/classes/janikowc/submitProject/submit_cs4280_P2` *`SubmitFileOrDirectory`*

Invocation:
`> P2[file]`
with `file` as before with implicit extension `.fs17`

Graded 90% execution 10% structure/standards.

Verify the grammar is LL(1) or rewrite as needed in an equivalent form.

Use your scanner module and fix if needed. If you fix any errors that you lost points for, ask to have some points returned after fixing

Implement the parser in a separate file (`parser.c` and `parser.h`) including the initial auxilary `parser()` function and all nonterminal functions. Call the parser function from main. The parser function generates error or returns the parse tree to main. In `testTree.c` (and `testTree.h`) implement a printing function using preorder traversal with indentations as before for testing purposes (2 spaces per level, print the node's label and any tokens from the node, then children left to right; one node per line). Call the printing function from main immediately after calling the parser and returning the tree. The printing function call must be later removed.

The project P2 will be tested assuming the simpler scanner - white spaces separate all tokens.

CFG

(Please ensure this uses only tokens detected in your P1, no exceptions)

<program> -> <vars> <block>
<block> -> **Begin** <vars> <stats> **End**
<vars> -> empty | **Var Identifier** <mvars>
<mvars> -> **.** | **,** **Identifier** <mvars>
<expr> -> <M> **+** <expr> | <M> **-** <expr> | <M>
<M> -> <F> **%** <M> | <F> * <M> | <F>
<F> -> **(** <F> **)** | <R>
<R> -> **[** <expr> **]** | **Identifier** | **Number**

```
<stats>     ->   <stat>  <mStat>
<mStat>     ->   empty |  <stat>  <mStat>
<stat>      ->   <in> | <out> | <block> | <if> | <loop> | <assign>
<in>        ->   Input Identifier ;
<out>       ->   Output <expr>  ;
<if>        ->   Check [ <expr> <RO> <expr> ] <stat>
<loop>      ->   Loop [ <expr> <RO> <expr> ] <stat>
<assign>    ->   Identifier : <expr>   ;
<RO>        ->   < | <= | >  | >= | ==  |  !=
```

Suggestions

First ensure the grammar is LL(1) or make it LL(1). Note that <expr> and <M> can be handled without rewriting (see class discussion)

Implement the parser in two iterations:

1. Starting without the parse tree. Have your parses generate error (line number and tokens involved) or print OK message upon successful parse.
   For each nonterminal, use a void function named after the nonterminal and use only explicit returns. Decide how to pass the token. Have the main program call the parser, after setting up the scanner if any.
   Be systematic: assume each function starts with unconsumed token (not matched yet) and returns unconsumed token. Use version control and be ready to revert if something gets messed up.
2. Only after completing and testing the above to satisfaction, modify each function to build a subtree, and return its root node. Assume each function builds just the root and connects its subtrees. Modify the main function to receive the tree built in the parser, and then display it (for testing) using the preorder treePrint().

Some hints for tree:

- every node should have a label consistent with the name of the function creating it (equal the name?)
- every function creates exactly one tree node (or possibly none)
- the number of children seems as 3 or 4 max but it is your decision
- all syntactic tokens can be thrown away, all other tokens (operators, IDs, Numbers) need to be stored
- when storing a token, you may need to make a copy depending on your interface

Test files for good programs

Create files using the algorithm to generate programs from the grammar, starting with simplest programs one different statement at a time and then building sequences of statements and nested statements. You may skip comments but then test comment in some files.

Examples. Start with simplest programs and no variables and no comments.

```
Begin
  Output 1 ;
End
```

```
Begin
  Input x ;
End
```

```
Begin
```

```
  x : y ;
End

--------------------
Begin
  Check [ 0 == 0 ]
    Output 0 ;
End

---------------------
Begin
  Loop [ 0 == 0 ]
    Output 0 ;
End
```

Now add variables e.g.

```
----------------------
Var x .
Begin
  Output 1 ;
End

----------------------
Var x , y .
Begin
  Output 1 ;
End
```

Now add 2 statements at a time.

Then add 3 statements at a time.

Then use some statements as blocks, and try without variables in the block and then with variables in the block. Etc.

```
---------------------------

Var x , y .
Begin
  Input x ;
  Output 2 ;
  Begin
    Var x1 , x2 ;
    x : 2 + 3 % y ;
  End
  Loop [ x + 2 <= y + [ 2 - 3] + ( y )  ]
  Begin
    Loop [ x == ( [ x ] ) ]
    Begin
      Var x2 ;
      Output 1 ;
    End
End
```

Test files for bad programs

Create some bad programs, possibly by modifying with an error each program that was good. Make sure the error is discovered.