Parser and Tree Builder

Total 120 points (80 for parser and 40 for tree builder)

/accounts/classes/janikowc/submitProject/submit\_cs4280\_P2 *SubmitFileOrDirectory*

Invocation:   
> frontEnd [*file*]  
with *file* as before with implicit extension **.sp18  
Wrong invocations may not be graded.**

Graded 90% execution 10% structure/standards.

Verify the project 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. IntestTree.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.

# **P2 BNF**

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

<program>  ->     **program** <vars> <block>  
<block>       ->      **start**<vars> <stats> **stop**  
<vars>          ->      empty | **var Identifier = Integer**<mvars>   
<mvars>     ->     **.**  | **:** **Identifier** <mvars>  
<expr>        ->      <M> **+** <expr> | <M> **-** <expr> | <M> **/** <expr> | <M> **\*** <expr> | <M>  
<M>              ->     **%** <M> |  <R>  
<R>              ->      **(** <expr>**)**| **Identifier** | **Number**    
<stats>         ->      <stat>  <mStat>  
<mStat>       ->      empty |  <stat>  <mStat>  
<stat>           ->      <in> | <out> | <block> | <if> | <loop> | <assign>  
<in>              ->      **read Identifier .**   
<out>            ->      **print**<expr> **.**  
<if>               ->      **iff (** <expr> <RO> <expr> **)** <stat>  
<loop>          ->     **iter (** <expr> <RO> <expr>**)** <stat>  
<assign>       ->      **let Identifier  =** <expr> **.**  
<RO>            ->      **<** | **<  <** | **>** | **>  > | =  =** |  **! =**

# **P2 Suggestions**

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

Note that the parser calls the scanner, but the parser may need some setup in the main.

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

# **P2 Testing**

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. Start with shortest simplest program, then more and more complex. Make sure to have sequences of statements, nested statements (blocks), nested ifs and loops, variables in various blocks, etc, and to test all operators.

Here are some example files (more to follow)

program  
start  
  print 1 .  
stop