P2: Parser and Tree Builder
Due April 8, 2018 at 11:59pm

- 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.cpp 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.cpp (and testTree.h) implement a printing function using preorder traversal with indentations 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.
- Project P2 will be tested assuming that white spaces separate all tokens.
- Invocation:

```
frontEnd [file]
```

    - Wrong invocations may not be graded

BNF

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

<S> -> **program** <vars> <block>
<block> -> **start** <vars> <stats> **end**
<vars> -> empty | **var Identifier** <vars>
<expr> -> <H> **+** <expr> | <H> **-** <expr> | <H> **/** <expr> | <H> ***** <expr> | <H>
<H> -> **#** <R> | <R>
<R> -> **(** <expr> **)** | **Identifier** | **Integer**
<stats> -> <stat> <mStat>
<mStat> -> empty | <stats>
<stat> -> <in> **,** | <out> **,** | <block> **,** | <ifstat> **,** | <loop> **,** | <assign> **,**
<in> -> **read Identifier**
<out> -> **print** <expr>
<ifstat> -> **if (** <expr> <O> <expr> **)** <stat>
<loop> -> **iter (** <expr> <O> <expr> **)** <stat>
<assign> -> **let Identifier** = <expr>
<O> -> **<** | **>** | **:**

Lexical Definitions (same as for P1)

- All case sensitive
- Alphabet
  - all English letters (upper and lower), digits, plus the extra characters as shown below, plus WS
  - No other characters allowed and they should generate lexical errors
  - Each scanner error should display "Scanner Error:" followed by details including line number
- Identifiers
  - begin with a *lower case* letter and
  - continue with any number of letters or digits
  - you may assume no identifier is longer than 8 characters
- Keywords (reserved, suggested individual tokens)
  - `start end iter void var return read print program if then let`
- Operators and delimiters group.
  - **= < > : + - * / # . ( ) , { } ; [ ]**
- Integers
  - any sequence of decimal digits, no sign
  - you may assume no number longer than 8 characters
- Comments start with **!** and end with **!**

P2 Suggestions

- Ensure the grammar is LL(1) or make it LL(1):
  - Note that <expr> 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:
  - Starting without the parse tree.
    - Have your parser 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.
  - 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().
- Idea for printing tree with indentations:

```
static void printParseTree(nodeType *rootP,int level) {
  if (rootP==NULL) return;
  printf("%*c%d:%-9s ",level*2,' ',level,NodeId.info); // assume
some info printed as string
  printf("\n");
  printParseTree(rootP->child1,level+1);
  printParseTree(rootP->child2,level+1);

...
}
```

- 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

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 is one example file:

```
program
start
read var1 ,
end
```

- Here is another example file:

```
program
var id1
var id2
start
let id1 = 30 ,
if ( id1 : 14 ) ,
  let id2 = # id1 ,
end
```

Grading

- 120 points (80 for parser and 40 for tree builder)
- Programming and architectural style: 20%
- Execution: 80%