Parser and Tree Builder

Assume the project grammar is LL(1) (let me know if you suspect problems). You need to compute First() sets as needed (for any nonterminal that has multiple productions). If a token doesnt match any of the sets in the function for the nonterminal, issue error; however, if there is the empty production then instead of error you use the production (return from the function).

Use your P1 scanner module and fix if needed. If you fix any errors that you lost substantial points for on P1, ask to have some points returned after P2 works (ask in office hours).

Implement the parser in a separate file (parser.c and parser.h) including the initial auxiliary parser() function and all nonterminal functions. Call the parser function from main after processing the main arguments and anything needed to initiate your scanner.

The parser function generates an error on a syntax error or otherwise returns the parse tree to main upon success. In testTree.c (and testTree.h) implement a printing function using preorder left to right traversal with indentations as in P0 for testing purposes . Call this printing function from main immediately after calling the parser, on traverse and print the returned tree.

BNF

Please ensure this uses only tokens detected in your P1, or a subset, let me know otherwise. The grammar may need some rewrite to be LL(1) but this is doable with very few manipulations, let me know if you see problems (I am still verifying).

```
<program>   ->    <vars> main <stats> end
<vars>      ->    empty | let <varList> .                 // added . 11/5/23
<varList>   ->    id = integer | id = integer <varList>
<exp>       ->    <M> + <exp> | <M> - <exp> | <M>
<M>         ->    <N> * <M> | <N>
<N>         ->    <R> / <N> | - <N> |   <R>
<R>         ->    [ <exp> ]   | id | integer
<stats>     ->    <stat>   <mStat>
<mStat>     ->    empty |   <stat>   <mStat>
<stat>      ->    <in>    | <out>    | <block> | <if>   | <loop>   | <assign>
<block>     ->    start <vars> <stats> stop
<in>        ->    scan identifier .
<out>       ->    print <exp> .
<if>        ->    cond ( <exp> <RO> <exp> ) <stat>
<loop>      ->    loop ( <exp> <RO> <exp> )   <stat>
<assign>    ->    id   ~ <exp> .
<RO>        ->    <= (one token)   | >=  (one token)   | < | > | = | ~
```

P2 Suggestions

Make sure to verify the BNF grammar is LL(1) or make any necessary changes (other than optional as noted below).

A <nonterminal> with common prefixes will not have disjoint First() sets, it is not LL(1). These can be left-factorized until LL(1), or they can be implemented as are without left-factorization using the delay as in the notes. For example, assuming
<exp> -> <M> * <exp> | <M> / <exp> | <M>
then the exp() function can be implemented just always calling M() and then checking which production to continue

```
  void exp () {
        call M() ; // always
        if (next token is /Tk) then assume you are processing "<M> / <exp>" and thus must continue with "/ <exp>"
        else if (next token is *Tk) then assume you are processing "<M> * <exp>" and thus must continue with "* <exp>"
        else return as it is the production <M>
```

Make sure to implement the parser() auxiliary function as in your notes, and then one function for each <nonterminal> in your grammar, naming each function the same as the <nonterminal>.

Once the grammar is LL(1), it means

1. Each <nonterminal> with multiple productions will have the First() sets disjoint to make the correct prediction looking at the next token from the scanner
2. If the token doesnt match any First() sets in this function (for this <nonterminal>) then
   - If the <nonterminal> has the empty production, use it (return from the function)
   - Otherwise, issue a detailed error and exit
3. If you are trying to match a token and they dont match, issue a detailed error message and exit

Note that the parser should be calling the scanner for each new token. This should happen always when (and if and only if) the current token in the production was matched with the token from the scanner.

Implement the parser in two iterations (as illustrated in the notes in two separate stages):

1. Starting without building the parse tree. Have your parses generate error (detailed, including involved tokens and their line number of processed) or print OK message upon successful parse.   The ok message will later be replaced with a call to back end processing.
   For each <nonterminal>, use a void function named after the <nonterminal> and use only explicit returns. Decide how to pass the token, could be a global variable in the file (static). Have the main program call the parser after all the preliminary work needed there.
   Be systematic: assume each function starts with unconsumed token (not matched yet) and returns unconsumed token (not matched). Use version control and be ready to revert if something gets messed up.
2. Only after completing and testing the above to a satisfaction, modify each function to build a subtree, and return its root node. Assume each function builds just its root node and connects its subtrees. Modify the main function to receive the tree built in the parser, and then display it (for testing only) using the preorder tree printing.

Some summary hints for tree:

- every node should have a label consistent with the name of the function creating it (can be string with function name)
- every function creates exactly one tree node (or possibly none if going into the empty production)
- the number of children seems as 3 or 4 max (depending on some choices, it is the maximum number of <nonterminal>s listed in any production) but it is your decision
- all syntactic tokens can be thrown away, all other tokens (operators in expressions, relational operators, identifiers, integers) need to be stored in the node processing them
- when storing a token, you may need to make a copy depending on your interface (is there just one global token or each token has separate memory)

Test files will be posted - good programs that should succeed in the parser, and bad programs that should throw specific errors. Make sure all good programs do not generate errors and all bad programs display the error message.

P2 Testing - 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 then nested statements, expressions, etc. You may skip comments but then test a comment in some files. Make sure to have sequences of statements, nested statements (blocks), nested ifs and loops, variables in various blocks, expressions, and to test all operators.

Here are some example files. **If you see potential problems please check with me. These programs should NOT generate scanner nor parser errors.**

The first tests should be on the parser only, no tree generation.Then the final test should do the same, and also observe the printed tree for testing to make sure that the tree is correct. Each tree needs to have all semantic tokens and the proper structure corresponding to the program.

Start with shortest program, test comments, test variables, the add progressively more complex programs.

```
main #shortestProgramPrint1#
  print 1 .
end







main
  print 1 .
end









let aa = 1
ab = 2
ac = 3 .
main
  print 1 .
end
```

```
main
  print 1 + 2 / 3 - 4 * 5 - [ 1 - - - 2 ] .
end
```

```
main
  print 1 .
  scan a .
  a ~ 2 + 3 .
  start
    let aa = 2 .
    print 1 .
  stop
end
```

```
main
  print 1 .
  start
    print 1 .
  stop
  start
    start
      print 1 .
    stop
  stop
end
```

```
main
  print 1 .
  cond ( 1 < 2 )
    print 1 .
end
```

```
main
  print 1 .
  cond ( 1 + 2 / 3 < 2 - 1 )
    cond ( 1 >= 2 )
      print 1 .
end
```

```
main
  print 1 .
  loop ( 1 < 2 )
    print 1 .
end





main
  print 1 .
  loop ( 1 + 2 / 3 < 2 - 1 )
    loop ( 1 >= 2 )
      print 1 .
end
```

P2 Testing - Bad Programs

Take one good program at a time and introduce a syntax error. Not lexical error. We do not have yet semantics to have semantics errors. There can be various kinds of errors: tokens added, deleted, or replaced with something that is invalid, and extra tokens after the end. Here are examples. On an error, a proper message should be displayed with line number (if using) and the parser should abort.

```
main #shortestProgramPrint1$
  print 1 .
end

main
  print 1 .
end
print 1 .


main
  let aa = 1 .
  print 1 .
end

main
  print 1 + 2 / 3 - 4 * + 5 - [ 1 - 2 ] .
end

main
  print 1 .
  scan a ;
  a ~ 2 + 3 .
  start
    aa = 2 .
    print 1 .
  stop
end

main
  print 1 .
  start
    print 1 .
  stop
  start
    start
      print 1 ;
    stop
  stop
end

main
  print 1 .
  cond ( 1 < 2 )
```

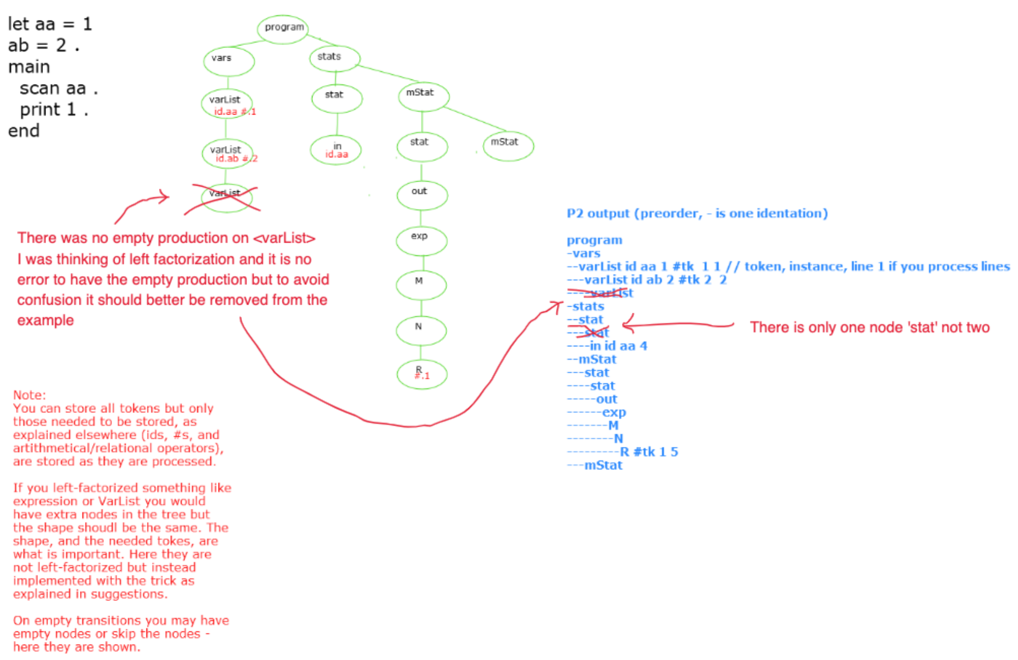
```
    print 1
end

main
  print 1 .
  cond ( 1 + 2 / 3 < 2 - 1 )
    cond ( 1 ~ - 2 )
      let a = 2 .
end

main
  print 1 .
  loop [ 1 < 2 ]
    print 1 .
end

main
  print 1 .
  loop ( 1 + 2 / 3 < 2 - 1 )
    loop ( 1 >= 2 )
      print 1
end
```

Tree Output Example



```
let aa = 1
ab = 2 .
main
  scan aa .
  print 1 .
end
```

There was no empty production on <varList>
I was thinking of left factorization and it is no
error to have the empty production but to avoid
confusion it should better be removed from the
example

Note:
You can store all tokens but only those needed to be stored, as explained elsewhere (ids, #s, and artithmetical/relational operators), are stored as they are processed.

If you left-factorized something like expression or VarList you would have extra nodes in the tree but the shape shoudl be the same. The shape, and the needed tokes, are what is important. Here they are not left-factorized but instead implemented with the trick as explained in suggestions.

On empty transitions you may have empty nodes or skip the nodes - here they are shown.

P2 output (preorder, - is one identation)

```
program
-vars
--varList id aa 1 #tk  1 1 // token, instance, line 1 if you process lines
---varList id ab 2 #tk 2  2
~~----varList~~
-stats
--stat
~~---stat~~
----in id aa 4
--mStat
---stat
----stat
-----out
------exp
-------M
--------N
---------R #tk 1 5
---mStat
```

There is only one node 'stat' not two