Parser and Tree Builder

Total 100 points.

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

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

Graded 90% execution 10% structure/standards.

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 scanner module and fix if needed. If you fix any errors that you lost points for on P1, ask to have some points returned after P2 works.

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 after all the checkes on the main arguments and anything needed to initiate your scanner.

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 this printing function from main immediately after calling the parser, on the returned tree. **The printing function call must be later removed for P3.**

The project P2 will be tested assuming white spaces separate all tokens.

# **P2 BNF**

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

<program>  ->     <vars> <block>  
<block>       ->      **begin**<vars> <stats> **end**  
<vars>         ->      empty | **data Identifier =  Integer  .**<vars>  
<expr>        ->      <N> **-** <expr>  | <N>  
<N>             ->       <A> / <N> | <A> \* <N> | <A>  
<A>             ->        <M> **+** <A> | <M>  
<M>              ->     **\*** <M> |  <R>  
<R>              ->      **(** <expr>**)**| **Identifier** | **Integer**  
<stats>         ->      <stat> <mStat>  
<mStat>       ->      empty |  <stat> <mStat>  
<stat>           ->      <in>**.**  | <out> **.** | <block>| <if> **.** | <loop> **.** | <assign> **.**   
<in>              ->      **in  Identifier**  
<out>            ->      **out**<expr>  
<if>               ->      **iffy [** <expr> <RO> <expr> **]** **then** <stat>  
<loop>          ->     **loop  [** <expr> <RO> <expr>**]**<stat>  
<assign>       ->      **Identifier  =** <expr>   
<RO>            ->      **<** | **<  <**(two tokens >)  | **>** | **>  >**(two tokens)**|  ==**(one token ==) |  **<  >**(two tokens)

# **P2 Suggestions**

Make sure to implement the parser() auxiliary function as in your notes, and then a function for each nonterminal in your grammar.

The grammar is LL(1), which 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 (this nonterminal) then
   * If the nonterminal has the empty production, use it (return from the function)
   * Otherwise, issue error and exit
3. Note that some nonterminals with common prefixes will not have disjoint First() sets but can be implemented using the delay as in the notes. For example, <expr> can be implemented as (could also be factored out of desired)

    void expr () {  
        call N() ; // always  
        if (next token is minusTk) then assume you are processing "<N> - <expr>" and thus must continue with "- <expr>"  
        else return

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 teh next token from the scanner.

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

1. Starting without 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 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. 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. 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 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) using the preorder tree printing.

 Some 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 nonterminals listed in any production) but it is your decision
* all syntactic tokens can be thrown away, all other tokens (operators, IDs, Numbers) 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)

Tests will be posted - good programs that should succeed in the parser, and bad programs that should throw specific errors.

# **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 or 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.

First group: shortest programs with one statement. replace print with one other statement at a time, including block as also shown.

begin  
 out 1 .  
end  
  
data x = 1 .  
data y = 2 .  
begin  
 out 1 .  
end  
  
data x = 1 .  
data y = 2 .  
begin  
 data z = 3 .  
 out 1 .  
end  
  
data x = 1 .  
data y = 2 .  
begin  
 data z = 3 .  
 begin   
 out 1 .  
 end   
 out 2 .  
end  
  
data x = 1 .  
data y = 2 .  
begin  
 in x .  
 out x + 2 / ( 2 - y ) .  
end  
  
data x = 1 .  
begin  
 in x .  
 iffy [ x < 0 ] then  
 out x . .  
end  
  
data x = 1 .  
begin  
in x .  
iffy [ x < 0 ] then  
 begin  
 x = 5 .  
 out x .  
 end .  
end  
  
data x = 1 .  
begin  
in x .  
loop [ x < 0 ]  
 out x . .  
end  
  
data x = 1 .  
begin  
in x .  
loop [ x < 0 ]  
 begin  
 x = 5 .  
 out x .  
 end .  
end  
  
data x = 1 .  
begin  
 in x .  
 iffy [ x < 0 ] then  
 iffy [ x < > 0 ] then   
 out x . . .  
end  
  
data x = 1 .  
begin  
in x .  
loop [ x + 2 / 3 < 0 - \* 3 ]  
 iffy [ x < > 0 ] then   
 out x . . .  
end

# **P2 Testing - Bad Programs**

Take one good program at a time and introduce an error. There can be two kinds of errors: token added, deleted, or replaced with something that is invalid, and extra tokens after the end. Here is an example. On an error, a proper message should be displayed and the parser should abort.

begin  
out 1 .  
end .  
  
data x = 1 .  
data y = 2 .  
begin  
 out 1 ;  
end  
  
data x = 1 .  
data y = 2 .  
begin  
 var z = 3 .  
 out 1 .  
end  
  
data x = 1 .  
data y = 2 .  
begin  
 data z = 3 .  
 begin   
 out 1 .  
 end .  
 out 2 .  
end  
  
data x = 1 .  
data y = 2 .  
begin  
 in x .  
 out x + - 2 / ( 2 - y ) .  
end  
  
data x .  
begin  
 in x .  
 iffy [ x < 0 ] then  
 out x .   
end  
  
data x = 1 .  
begin  
 in x .  
 iffy [ x < 0 ] then  
 begin  
 x = 5 .  
 out x .  
 end   
end  
  
data x = 1 .  
begin  
 in x .  
 loop [ x + 2 / 3 < 0 - \* 3 ]  
 iffy [ x < > 0 ] then   
 out x . .   
end