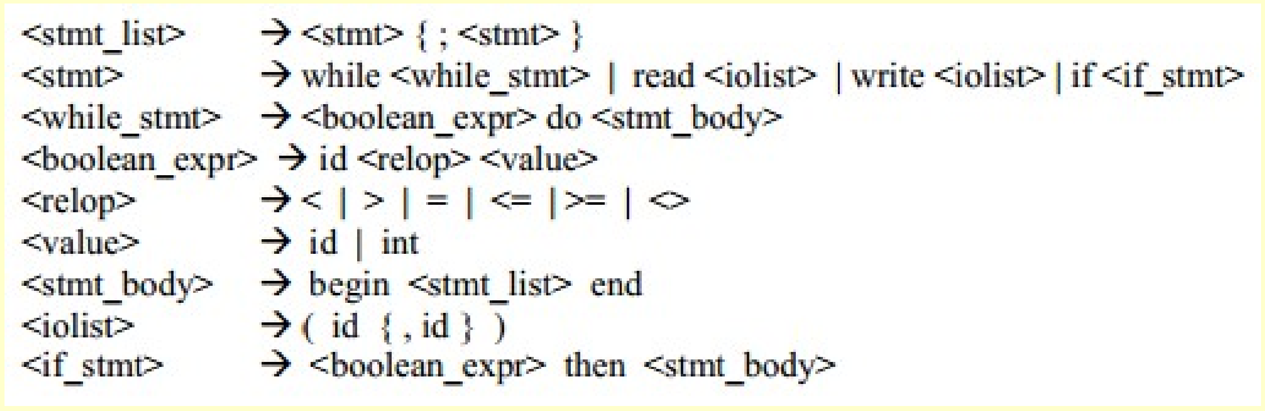
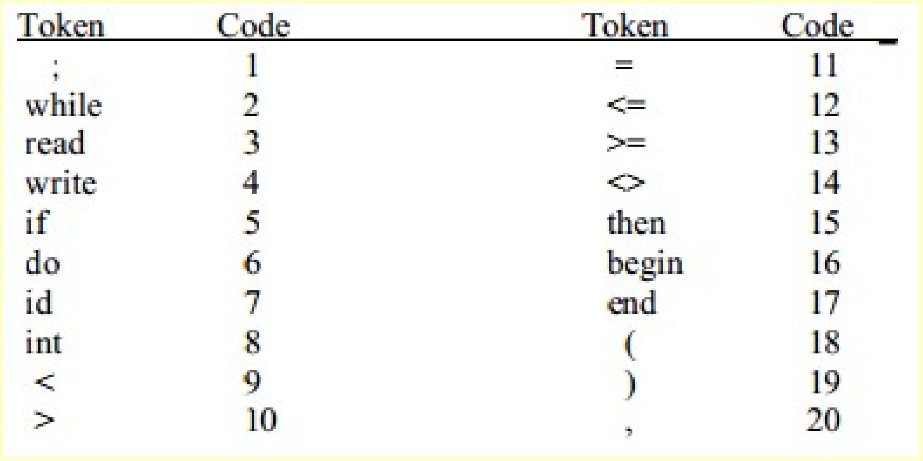
Project : Building a Syntax Analyzer in Java

Write a recursive descent parser that will parse statement lists using the grammar:



```
<stmt_list>       → <stmt> { ; <stmt> }
<stmt>            → while <while_stmt> | read <iolist> | write <iolist> | if <if_stmt>
<while_stmt>      → <boolean_expr> do <stmt_body>
<boolean_expr>    → id <relop> <value>
<relop>           → < | > | = | <= |>= | <>
<value>           → id | int
<stmt_body>       → begin <stmt_list> end
<iolist>          → ( id { , id } )
<if_stmt>         → <boolean_expr> then <stmt_body>
```

Input to the parser will be a stream of tokens (these are the terminals in the grammar) represented by the following code:



Token	Code	Token	Code
;	1	=	11
while	2	<=	12
read	3	>=	13
write	4	<>	14
if	5	then	15
do	6	begin	16
id	7	end	17
int	8	(	18
<	9	)	19
>	10	,	20

For example: A statement of the form: if a = b then write (c) would be represented as

5 7 11 7 15 4 18 7 19

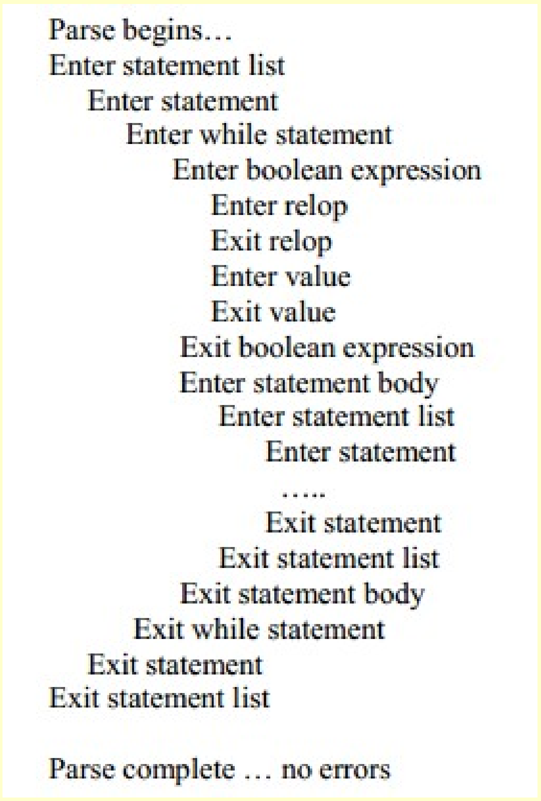
(These numbers may or may not be on the same line in the input file.)

Details:

1. Write a function for each variable (nonterminal) in the grammar. The functions should return (either through a parameter or as a return value) a boolean value to indicate whether the parse was successful or not. Each function should have available the current token (generally as a parameter).

2. Write a driver that contains any initializations necessary, calls "gettoken" to get the first token (note – in this simplified version, all gettoken does is read in the code – in a "real" parser, gettoken would be a lexical analyzer that would read the input and send a code indicating the token), then calls the function for <stmt_list> (this is the start symbol in this grammar for statement lists). The driver should print a message about the success of the parse or about the number of errors encountered.

3. Print out a trace of the parse with indentation to indicate the levels within the parse. Do this by putting print statements at the beginning and end of each function. For example, a parse may look like the following:



```
Parse begins…
Enter statement list
    Enter statement
        Enter while statement
             Enter boolean expression
                 Enter relop
                 Exit relop
                 Enter value
                 Exit value
              Exit boolean expression
              Enter statement body
                  Enter statement list
                       Enter statement
                         …..
                       Exit statement
                  Exit statement list
              Exit statement body
         Exit while statement
    Exit statement
Exit statement list

Parse complete … no errors
```

4. Error recovery: When a specific terminal symbol is expected and is not found, print a message stating which symbol is expected, set error flags and an error count, then recurs back to the function for statement list (this should be a natural result of your if … then … else's). In statement list, call a function to flush the input to the next semicolon (call gettoken until the next semicolon is encountered (that is ignore all tokens remaining in the statement you were trying to parse)). (In some cases, you may not want to go all the way back to statement list – you can pick the parse back up earlier by writing a flush function that flushes to a token you specify in the parameter list.)

Additional Requirements:

- The program must be written in Java. You should zip all the files including README and submit it to D2L

- Input will be from a file. Your program should prompt for the file name.

- Your program must be documented appropriately and use good programming practices