Consider the following EBNF grammar for a very simple Java Class definition in the Java programming language:

Note: varlist, assignstatemt (and varname in returnstatemt) are followed by a ";".

<javaclass> ::= <classname> [X <classname>] B <varlist>; {<method>} E
<classname> ::= C|D
<varlist> ::= <vardef> {, <vardef>}
<vardef> ::= <type> <varname> | <classname> <varref>
<type> ::= I|S
<varname> ::= <letter> {<char>}
<letter> ::= Y|Z
<char> ::= <letter> | <digit>
<digit> ::= 0|1|2|3
<integer> ::= <digit> {<digit>}
<varref> ::= J|K
<method> ::= <accessor> <type> <methodname> ([<varlist>]) B {<statemt>} <returnstatemt> E
<accessor> ::= P|V
<methodname> ::= M|N
<statemt> ::= <ifstatemt> | <assignstatemt>;|<whilestatemt>|<methodcall>
<ifstatemt> ::= F <cond> T B {<statemt>} E [L B {<statemt>} E]
<assignstatemt> ::= <varname> = <mathexpr> | <varref> = <getvarref>
<mathexpr> ::= <factor> {+ factor}
<factor> ::= <oprnd> {* oprnd}
<oprnd> ::= <integer> | <varname> | (<mathexpr>) | <methodcall>
<getvarref> ::= O <classname>() | <methodcall>
<whilestatemt> ::= W <cond> T B {<statemt>} E
<cond> ::= (<oprnd> <operator> <oprnd>)
<operator> ::= < | = | > | !
<returnstatemt> ::= R <varname>;
<methodcall> ::= <varref>.<methodname>([<varlist>])

KEY: The single letters are codes after lexical analysis for the following words:
The tokens are:
X for extends
B for Begin of block
E for End of block
I for Integer
S for String
P for public
V for private
F for if
T for then
L for else

O for new (to create a new class Object reference)
W for while
R for return

```
The tokens also include:  C D I S P V 0 1 2 3 J K M N ( ) + * < = > !
Nonterminals are shown as lowercase words.
The following characters are NOT tokens (they are EBNF metasymbols):   | { } [ ]
Note that parentheses are TOKENS, not EBNF metasymbols in this particular grammar.

1. Draw Syntax Diagrams for the above grammar.

2. Show that the grammar satisfies the two requirements for predictive parsing.
   (it does, you just need to prove it).

3. Implement a recursive-descent recognizer:
    - Prompt the user for an input stream.
    - The user enters an input stream of legal tokens, followed by a $.
    - You can assume:
        the user enters no whitespace,
        the user only enters legal tokens listed above,
        the user enters one and only one $, at the end.
    - The start symbol is *function* (as defined above)
    - Your program should output "legal" or "errors found" (not both!).
        You can report additional information as well, if you want.
        For example, knowing where your program finds an error could
        be helpful for me to assign partial credit if it's wrong.
    - Assume the input stream is the TOKEN stream.  Assume that any
        whitespace has already been stripped out by the lexical scanner.
        (i.e., each token is one character - lexical scanning has been completed)
    - Since the incoming token stream is terminated with a $,
        you will need to add the $ to the grammar and incorporate it
        in your answers to questions #1 and #2 above, where appropriate.
    - Use Java, C, or C++, or ask your instructor if you wish
        to use another language.
    - Limit your source code to ONE file.
    - Make sure your program works on ATHENA before submitting it.
    - INCLUDE INSTRUCTIONS FOR COMPILING AND RUNNING YOUR PROGRAM ON ATHENA
        IN A COMMENT BLOCK AT THE TOP OF YOUR PROGRAM.  Also, explain any
        input formatting that your program requires of the user entry.

4. Collect the following submission materials into ONE folder:
    - your source code file
    - syntax diagrams (may be handwritten/scanned, or computer drawn)
    - proof of predictive parsing (may be handwritten/scanned, or computer drawn)

5. ZIP the one folder into a single .zip file.

6. Submit your ONE .zip file to the dropbox in Canvas.
```