Compiler Design Lab

Consider the following mini Language, a simple procedural high-level language, only operating on integer data, with a syntax looking vaguely like a simple C crossed with Pascal. The syntax of the language is defined by the following BNF grammar:

```
<program> ::= <block>

<block> ::= { <variabledefinition> <slist> }
        | { <slist> }

<variabledefinition> ::= int <vardeflist> ;

<vardeflist> ::= <vardec> | <vardec> , <vardeflist>

<vardec> ::= <identifier> | <identifier> [ <constant> ]

<slist> ::= <statement> | <statement> ; <slist>

<statement> ::= <assignment> | <ifstatement> | <whilestatement>
        | <block> | <printstatement> | <empty>

<assignment> ::= <identifier> = <expression>
        | <identifier> [ <expression> ] = <expression>

<ifstatement> ::= if <bexpression> then <slist> else <slist> endif
        | if <bexpression> then <slist> endif

<whilestatement> ::= while <bexpression> do <slist> enddo

<printstatement> ::= print ( <expression> )

<expression> ::= <expression> <addingop> <term> | <term> | <addingop>
<term>

<bexpression> ::= <expression> <relop> <expression>

<relop> ::= < | <= | == | >= | > | !=

<addingop> ::= + | -
```

$\langle multop\rangle ::= * \mid /$

$\langle factor\rangle ::= \langle constant\rangle \mid \langle identifier\rangle \mid \langle identifier\rangle [\ \langle expression\rangle]$

$\qquad \mid (\ \langle expression\rangle)$

$\langle constant\rangle ::= \langle digit\rangle \mid \langle digit\rangle \langle constant\rangle$

$\langle identifier\rangle ::= \langle identifier\rangle \langle letterordigit\rangle \mid \langle letter\rangle$

$\langle letterordigit\rangle ::= \langle letter\rangle \mid \langle digit\rangle$

$\langle letter\rangle ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z$

$\langle digit\rangle ::= 0|1|2|3|4|5|6|7|8|9$

$\langle empty\rangle$ has the obvious meaning

Comments (zero or more characters enclosed between the standard C/Java-style comment brackets /*...*/) can be inserted.   The language has rudimentary support for 1-dimensional arrays. The declaration int a[3] declares an array of three elements, referenced as a[0], a[1] and a[2]. Note also that you should worry about the scoping of names.

A simple program written in this language is:

```
{
int a[3],t1,t2;
 t1=2;
a[0]=1;
a[1]=2;
 a[t1]=3;
 t2=-(a[2]+t1*6)/(a[2]-t1);
if t2>5
then print(t2);
 else {
 int t3;
t3=99;
 t2=-25;
 print(-t1+t2*t3); /* this is a comment on 2 lines */
 }
endif }
```

1. Identify the valid tokens i.e., identifiers, keywords, operators etc., in the given language according to above BNF grammar.
2. Write a program for lexical analyzer to recognize all the valid tokens in the input program written according to the grammar.
3. Write a lex program to print the copy of input.
4. Write a lex program to display the number of lines and number of characters in input.
5. Write a lex program to replace the sequence of white spaces by a single blank from input.

6. Write a lex program to replace the sequence of white spaces by a single blank from input text file and add the contents in output text file.
7. Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Simulate the same in C language.
8. Implement the lexical analyzer using JLex, flex, flex or lex or other lexical analyzer generating tools.