Assignment: 3

var1.l

```
%{
#include<stdio.h>
#include "y.tab.h"
%}
letter          [A-Za-z]
digit           [0-9]
keyword         int|float|char
identifier      {letter}({letter}|{digit})*
%%
{keyword}       return TYPE;
","             return COMMA;
";"             return SC;
{identifier}    return ID;
"\n"            return NL;
%%
```

var1.y

```
%{
#include<stdio.h>
int yylex();
int yyerror();

%}
```

```
%token ID TYPE SC NL COMMA
%%
start:TYPE varlist SC NL    {printf("valid declarative statement");}
;

varlist:varlist COMMA ID
        |ID
;

%%
int yyerror()
{
printf("Invalid declarative statement");
}

int yywrap()
{
return 1;
}

int main()
{
yyparse();
}
```

Output: lex var1.l

yacc -d var1.y

cc lex.yy.c y.tab.c

./a.out

Explanation : This program consists of two parts: a Lex file (`var1.l`) and a Yacc file (`var1.y`). Together, they create a simple compiler that parses variable declarations in a language similar to C. Here's a detailed explanation of each part along with relevant theory.

Lex File (`var1.l`)

Lex (Flex) is a tool for generating scanners, which are programs that recognize lexical patterns in text.

Definitions Section

```c
%{
#include<stdio.h>
#include "y.tab.h"
%}
```

- `%{ %}`: C code inside these delimiters is copied verbatim into the generated C file. Here, we include the standard I/O header and the header generated by Yacc (`y.tab.h`).

```c
letter [A-Za-z]

digit  [0-9]

keyword  int|float|char

identifier {letter}({letter}|{digit})*
```

- `letter`: Matches any uppercase or lowercase letter.
- `digit`: Matches any digit.
- `keyword`: Matches any of the keywords `int`, `float`, or `char`.
- `identifier`: Matches a sequence starting with a letter followed by any number of letters or digits.

Rules Section

```c
%%
```

Marks the beginning of the rules section where patterns are associated with actions.

```c
{keyword}  return TYPE;
","       return COMMA;
";"       return SC;
{identifier} return ID;
"\n"      return NL;
```

- `{keyword}`: When a keyword is matched, return the token `TYPE`.
- `","`: When a comma is matched, return the token `COMMA`.
- `";"`: When a semicolon is matched, return the token `SC`.
- `{identifier}`: When an identifier is matched, return the token `ID`.
- `"\n"`: When a newline character is matched, return the token `NL`.

Yacc File (`var1.y`)

Yacc (Yet Another Compiler Compiler) is a tool for generating parsers, which are programs that recognize syntactic patterns in text.

Definitions Section

```c
%{
#include<stdio.h>
int yylex();
```

int yyerror();

%}

- `%{ %}`: C code inside these delimiters is copied verbatim into the generated C file. Here, we include the standard I/O header and declare `yylex` (generated by Lex) and `yyerror`.

```c
%token ID TYPE SC NL COMMA
```

- `%token`: Defines the tokens returned by the Lex scanner.

Grammar Rules Section

```c
%%
```

Marks the beginning of the grammar rules section.

```c
start: TYPE varlist SC NL {printf("valid declarative statement");}
;
```

varlist: varlist COMMA ID

      | ID

;

- `start`: The start symbol of the grammar.

  - `TYPE varlist SC NL`: A valid declarative statement must start with a `TYPE`, followed by a `varlist`, a semicolon (`SC`), and a newline (`NL`). If this pattern is matched, print "valid declarative statement".

- `varlist`: Defines a list of variables.

  - `varlist COMMA ID`: A `varlist` can be extended with an identifier (`ID`) preceded by a comma (`COMMA`).

  - `ID`: A `varlist` can also be a single identifier.

Additional C Code Section

```c
%%
```

Marks the end of the grammar rules section. The following code is the user code section.

```c
```

```
int yyerror()
{
  printf("Invalid declarative statement");
}

int yywrap()
{
  return 1;
}

int main()
{
  yyparse();
}
```

- `int yyerror()`: Defines the error handling function. If a syntax error is encountered, it prints "Invalid declarative statement".

- `int yywrap()`: Called when the end of the input is reached. Returning 1 indicates no more input.

- `int main()`: The main function which calls `yyparse()` to start the parsing process.

Summary

- **Lex file (`var1.l`)**: Scans the input and identifies tokens like `TYPE`, `COMMA`, `SC`, `ID`, and `NL`.

- **Yacc file (`var1.y`)**: Uses these tokens to parse and validate declarative statements in the form of `TYPE varlist ; \n`.

When the program is run, it reads input, the Lex scanner generates tokens, and the Yacc parser uses these tokens to determine if the input conforms to the defined grammar rules for valid declarative statements. If the input matches the grammar, it prints "valid declarative statement"; otherwise, it prints "Invalid declarative statement".