ASSIGNMENT B-4

TITLE: YACC program to validate variable declarations.

PROBLEM STATEMENT:

Write a program using YACC specifications to implement syntax analysis phase of compiler to validate type and syntax of variable declaration in Java.

OBJECTIVE:

- Be proficient on writing grammars to specify syntax
- Understand the theories behind different parsing strategies-their strengths and limitations
- Understand how the generation of parser can be automated
- Be able to use YACC to generate parsers

S/W PACKAGES AND HARDWARE REQUIREMENTS:

- 64-bit open source Linux (Fedora 20)
- Eclipse IDE, JAVA
- 64-bit architecture I3 or I5 machines
- LEX and YACC

OUTCOME:

We will be able to

- Match the variables and identify the data type.

THEORY:

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.
The following represents a simple pattern, composed of a regular expression that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

letter (letter | digit)*

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the "*" operator
- alternation, expressed by the "|" operator
- concatenation

Regular expressions are used for pattern matching.
Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline) and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements, enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, "**.**" and "**\n**", with an **ECHO** action associated for each pattern. Several macros and variables are predefined by lex. **ECHO** is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, **ECHO** is defined as:

#define ECHO fwrite(yytext, yyleng, 1, yyout)

Variable **yytext** is a pointer to the matched string (NULL-terminated) and **yyleng** is the length of the matched string. Variable **yyout** is the output file and defaults to stdout. Function **yywrap** is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a **main** function. In this case we simply call **yylex** that is the main entry-point for lex. Some implementations of lex include copies of **main** and **yywrap** in a library thus eliminating the need to code them explicitly. This is why our first example, the shortest lex program, functioned properly.

Here is a program that does nothing at all. All input is matched but no action is associated with any pattern so there will be no output.

```
%%
.
\n
```

The following example prepends line numbers to each line in a file. Some implementations of lex predefine and calculate **yylineno**. The input file for lex is **yyin** and defaults to **stdin**.

```
%{
int yylineno;
%}
%%
^(.*)\n printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
yyin = fopen(argv[1], "r");
yylex();
fclose(yyin);
}
```

The definitions section is composed of substitutions, code, and start states. Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with "**%{**" and "**%}**" markers. Substitutions simplify pattern-matching rules. For example, we may define digits and letters:

```
digit [0-9]
letter [A-Za-z]
%{
int count;
%}
%%
/* match identifier */
{letter}({letter}|{digit})* count++;
%%
int main(void) {
yylex();
printf("number of identifiers = %d\n", count);
return 0;
}
```

Whitespace must separate the defining term and the associated expression. References to substitutions in the rules section are surrounded by braces (**{letter}**) to distinguish them from literals. When we

have a match in the rules section the associated C code is executed. Here is a scanner that counts the number of characters, words, and lines in a file (similar to Unix wc):

```
%{
int nchar, nword, nline;
%}
%%
\n { nline++; nchar++; }
[^ \t\n]+ { nword++, nchar += yyleng; }
. { nchar++; }
%%
int main(void) {
yylex();
printf("%d\t%d\t%d\n", nchar, nword, nline);
return 0;
}
```

CONCLUSION:

Thus, we successfully implemented Lexical Analysis to count number of words, lines and characters.