**LEX:**

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions.

Here we even skip the **comments** :-

We use regular expressions to recognize the comments .

**Single line comment**

This one is easy because in flex, . won't match a newline. So the following will match from // to the end of the line, and then do nothing

"//".\* { /\* Do Nothing \*/ }

**Multiline comment**

The fact that \* is a regular expression character as well as a key part of the comment marker makes the following regex a bit hard to read. We use [\*] as a pattern which recognizes the character \*; in flex Essentially, the regular expression matches sequences of characters ending with a (string of) \* until it finds one where the next character is a /. In other words, it has the same logic as your C code.

[/][\*][^\*]\*[\*]+([^\*/][^\*]\*[\*]+)\*[/] { /\* DO NOTHING \*/ }

The above requires the terminating \*/; an unterminated comment will force the lexer to back up to the beginning of the comment and accept some other token, usually a / division operator. That's likely not what you want, but it's not easy to recover from an unterminated comment since there's no really good way to know where the comment should have ended. Consequently , we need to add an error rule:

[/][\*][^\*]\*[\*]+([^\*/][^\*]\*[\*]+)\*[/] { /\* DO NOTHING \*/ }

[/][\*] { fatal\_error("Unterminated comment"); }

Then we identify the keywords like int , char , if , else , while ,and , not , or , void , return & operators such as " \ [ ] ^ - ? . \* + | ( ) $ / { } % < >

Then we identify the literals and identifiers , the rule to identify an identifier is

{L}{A}\* where L [a-zA-Z\_] & A [a-zA-Z\_0-9] . After we identify the identifier , we make its entry into symbol table here it is mytable.h

We need to ignore the white spaces and the escape characters such as \t\v\n\f .

If we come across \n we need skip that line and increment the line number .

**YACC:**

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

**List of Tokens:**

* PAR\_OPEN
* PAR\_CLOSE
* COMMA
* SEMICOLON
* WHILE
* FOR
* DO
* RETURN
* IF
* ELSE
* ELSEIF
* CB\_OPEN
* CB\_CLOSE
* PLUS
* MINUS
* ASTERISK
* SLASH
* ASSIGNMENT
* OR
* AND
* NOT
* LESS
* LESS\_EQUAL
* MORE\_EQUAL
* MORE
* EQUAL
* NOT\_EQUAL
* QUOT
* NUMBER
* LITERAL\_C
* ID
* CHAR
* INT

**Types:**

* Expression
* char\_expression
* conditions
* types

**Grammar:**

* **program :** program funcdef | funcdef ;
* **funcdef :** types ID args block\_statement ;
* **args :** PAR\_OPEN var\_def\_list PAR\_CLOSE ;
* **var\_def\_list :** var\_def COMMA var\_def | var\_def | ;
* **var\_def :** types ID ;
* **types :** INT | CHAR | VOID;
* **block\_statement :** CB\_OPEN statements CB\_CLOSE ;
* **statements :** statements statement | statement ;
* **statement :** block\_statement | conditional\_statement | while\_st | for\_st | do\_while\_st | assignment\_statement SEMICOLON | ret\_statement SEMICOLON ;
* **conditional\_statement :** IF PAR\_OPEN conditions PAR\_CLOSE block\_statement elsest | IF PAR\_OPEN conditions PAR\_CLOSE block\_statement elseif ;
* **elsest :** ELSE block\_statement ;
* **elseif :** ELSEIF PAR\_OPEN conditions PAR\_CLOSE block\_statement elsest | ELSEIF PAR\_OPEN conditions PAR\_CLOSE block\_statement elseif ;
* **while\_st :** WHILE PAR\_OPEN conditions PAR\_CLOSE block\_statement
* **do\_while\_st:** DO block\_statement WHILE PAR\_OPEN conditions PAR\_CLOSE SEMICOLON
* **for\_st :** FOR PAR\_OPEN assignment\_statement SEMICOLON conditions SEMICOLON assignment\_statement PAR\_CLOSE block\_statement
* **conditions :** conditions LESS expression | conditions LESS\_EQUAL expression | conditions MORE\_EQUAL expression | conditions MORE expression | conditions NOT\_EQUAL expression | conditions EQUAL expression | expression ;
* **assignment\_statement :** types ID ASSIGNMENT expression | ID ASSIGNMENT expression | types ID ASSIGNMENT char\_expression | ID ASSIGNMENT char\_expression | types ID ASSIGNMENT | error ;
* **ret\_statement :** RETURN expression;
* **expression :** NUMBER | ID | expression PLUS expression | expression MINUS expression | expression ASTERISK expression | expression SLASH expression | PAR\_OPEN expression PAR\_CLOSE
* **char\_expression :** QUOT LITERAL\_C QUOT ;

**Syntax analysis**

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

**Semantic analysis ->**

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

while\_st : WHILE PAR\_OPEN conditions {removetemp(tempnum);} PAR\_CLOSE

{

char \* startwhile = openlabel();

char \* endwhile = openlabel();

stack\_push(endwhile);

stack\_push(startwhile);

printf("%s :\nif not %s goto %s ; \n",startwhile , $3, endwhile);

}

Here, stack is used for various purposes. Openlabel and closelabel are functions defined to assign label when a loop or conditional statement starts. Use of labels helps to redirect loop in situation where it completes the statements, that is it is similar to goto statements.

Push or pop operations are used for computing values of expression which is similar to computing results of infix and postfix expression.

conditions LESS expression { if(**istemp($1)**){

if(**istemp($3)**){**removetemp(1)**;}

printf("%s = %s < %s ;\n", $$ = $1 ,$1 , $3 ) ;

}

else if(istemp($3)){

if(istemp($1)){removetemp(1);}

printf("%s = %s < %s ;\n", $$ = $3 ,$1 , $3 ) ;

}

else{

printf("%s = %s < %s ;\n", $$ = newtemp() ,$1 , $3 ) ;

} }

Here, temp is nothing but temporary variable which is pushed and popped into stack for computation. Its is assumed that 3 operators can be computed in a stack.

Say, 3 + 4

Here, 3 -> $1

+ -> $2

4 -> $3

This elements are pushed into stack then undergo syntax and semantic analysis like shown above.

**// Create Temporary variable ( Ti )**

char \* newtemp(){

tempnum = tempnum + 1;

char integer\_string[4] = "";

sprintf(integer\_string, "%d", tempnum);

char \* temp ;

temp = strdup("T");

return strcat(temp, integer\_string);

}

**Assembler:**

The assembler translates the intermediate code to machine understandable code.

Python language is used for implementation of Assembler.

Its operation is as follows:

The programs read the 3 Address Code line by line and splits it into tokens with space as a delimiter. Depending on the number of tokens in a line and the sequence of tokens, the program identifies whether it is an

* Assignment Operator
* Relational Operator
* Boolean Operator
* Logical Operator

The code assigns op code specified for the identified operator.

Example: T1 = T1 + T2 ;

Here the token count is 6.

The code thus identifies it to be an assignment operator. The fourth token being a plus operator, therefore the machine opcode generated for this line is :

ADD T1,T2