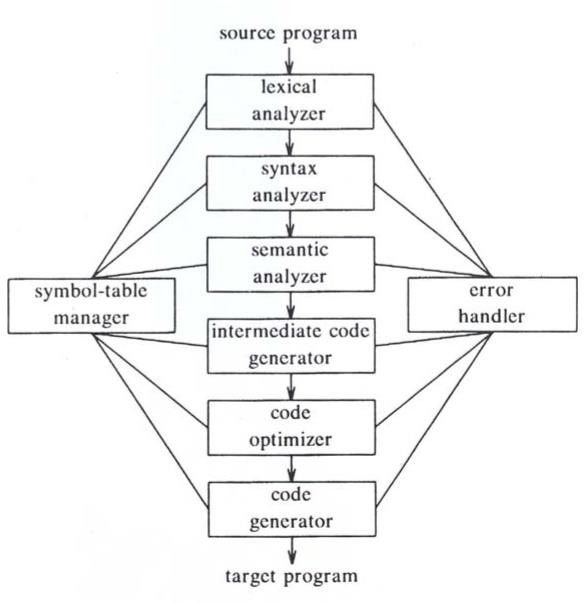
CS305 – Compiler Design Lab

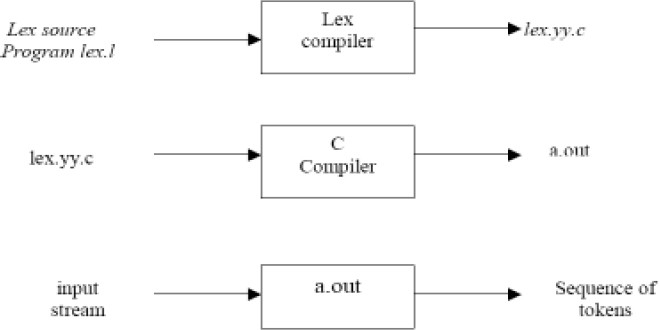
(11 August 2021 2-4 PM)

|  |  |
| --- | --- |
| Weekly Exercises | 20% |
| Mid Sem Exam | 15% |
| Project | 50% |
| End Sem Exam | 15% |

* 2 members in a team
* Lex and Yacc tool need to be installed in your PC
* Project consists of 4 phases:
  + Evaluation deadlines are given in course plan
* Except evaluation week, exercises will be given at the beginning of the lab. Students are expected to complete the exercises within the given lab hours and submit the same in IRIS.



Your task is to design a mini- compiler consisting of lexical, syntax, semantics and ICG phases for C programming language.

* LEX is a tool used to generate a lexical analyzer.
* LEX generates C code for a lexical analyzer, or scanner.
* It uses patterns that match strings in the input and converts the strings to tokens.

**Steps for execution:**

lex filename.l cc lex.yy.c

./a.out < inputfile

Lex input file is divided in to three parts:

/\* declarations \*/

…

%%

/\* rules \*/

....

%%

/\* user routines\*/

....

* + Declarations Section:
    - It contains declarations of simple name definitions to simplify the scanner specification
    - Definitions have the form:

name definition

* + - Example:

DIGIT [0-9]

ID [a-z][a-zA-Z0-9]\*

* + Rules Section:
    - It contains a series of rules of the form:

pattern action

* + - Example:

{ID} printf(“An identifier %s\n”,yytext);

* + User routine section:
    - They are auxiliary procedures needed by the actions.

// Lex program to count the number of identifiers

%{#include<stdio.h> **//Declarations**

int count=0; char ch=0;

%}

digit[0-9] letter[a-zA-Z\_]

%% **//Rules**

{letter}({letter}|{digit})\* { count++;

}

%%

int main() **//User Routines**

{

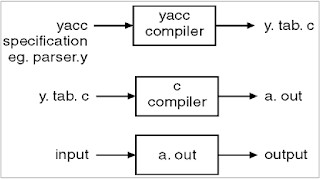
yylex();

printf("count: %d",count); return 0;

}

**Output:**

count:4

* + YACC is a tool to generate the parser
  + A parser is a program that checks whether its input (viewed as a stream of tokens) meets a given grammar specification.

YACC input file is divided in to three parts:

/\* declarations \*/

…

%%

/\* rules \*/

....

%%

/\* user routines \*/

....

* + Declaration section:

/\* Beginning of Declarations part \*/

%{

/\*Beginning of C declarations\*/

/\*End of C declarations\*/

%}

/\*Beginning of YACC declarations \*/

/\*End of YACC declarations \*/

/\* End of Declarations Part \*/

%%

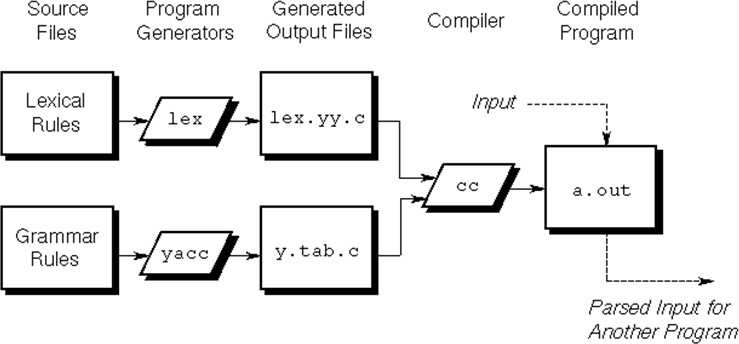
* + - It consists of two parts:
      * C declarations
      * YACC declarations
        + Tokens returned by the lexical analyzer
  + Rules Section:
    - It contains a series of productions of the form:

production action

* + - Example:

expr: DIGIT {printf("NUM%d ",pos);}

* + User routine section:
    - It contains the definitions of three mandatory functions main(), yylex() and yyerror().



**Execution of Lex and Yacc Programs:**

lex lexfile.l yacc yaccfile.y

cc lex.yy.c y.tab.h –ll

./a.out < inputfile

# Sample Lex and Yacc Program

**Lex file: calc.l**

%{

#include "y.tab.h" extern int yylval;

%}

**//Declarations**

%%

[0-9]+ { yylval=atoi(yytext); **//Rules**

return NUMBER;}

\n {return 0;}

. {return yytext[0];}

%%

Yacc file: calc.y

%{

#include<stdio.h> int yylex(void);

int yyerror(char \*);

%}

%token NAME NUM

%%

statement : NAME '=' exp

| exp { printf("=%d\n",$1);}

;

exp:exp'+' NUM{$$ = $1+$3;}

|exp '-' NUM {$$ = $1-$3;}

|exp '\*' NUM {$$ = $1\*$3;}

**//Declarations**

**//Rules**

|exp '/' NUM { if ($3!=0){ $$ = $1/$3; }else { printf("Error: divide by Zero"); } }

|NUM {$$=$1;}

;

%%

int main()

{

yyparse(); return 0;

}

int yyerror(char \*s)

{

printf("%s",s);

}

**//User Routines**

**1. 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**: ::= ::= { } | { } ::= int ; ::= | , ::= | [ ] ::= | ; ::= | | | | | ::= = | [ ] = ::= if then else endif | if then endif ::= while do enddo ::= print ( ) ::= | | ::= ::= < | <= | == | >= | > | != ::= + | - ::= | ::= \* | / ::= | | [ ] | ( ) ::= | ::= | ::= | ::= 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 ::= 0|1|2|3|4|5|6|7|8|9 has the obvious meaning Comments (zero or more characters enclosed between the standard C/Javastyle 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.