**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

|  |  |  |
| --- | --- | --- |
| **Program No: 1** | **Roll Number:** 23071A0573 | **Date: 23|07|25** |
| **Program Title:** Lexical Analyzer | | |

# Aim:

To design a lexical analyzer

# Description:

Design a Lexical analyzer for the above language. The lexical analyzer should ignore redundant spaces, tabs and newlines. 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.

# Program:

#include <iostream> #include <fstream> #include <regex> #include <string> #include <vector> #include <set> #include <iterator>

using namespace std;

// List of C/C++ keywords (including data types) set<string> C\_CPP\_KEYWORDS = {

"auto", "break", "case", "char", "const", "continue", "default", "do",

"double", "else", "enum", "extern", "float", "for", "goto", "if",

"int", "long", "register", "return", "short", "signed", "sizeof",

"static", "struct", "switch", "typedef", "union", "unsigned", "void",

"volatile", "while", "main", "namespace", "public", "private", "protected",

"class", "friend", "virtual", "template", "this", "new", "delete", "try",

"catch", "throw", "inline", "using", "export", "mutable"

};

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

// Regular expressions for various components const regex

KEYWORD\_PATTERN(R"(\b(?:auto|break|case|char|const|continue|default|do|double|else|enum|exte rn|float|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned| void|volatile|while|main|namespace|public|private|protected|class|friend|virtual|template|this|new|delet e|try|catch|throw|inline|using|export|mutable)\b)");

const regex IDENTIFIER\_PATTERN(R"(\b[A-Za-z\_][A-Za-z0-9\_]\*\b)"); const regex OPERATOR\_PATTERN(R"([+\-\*/%=<>!&|^~]|\\*\\*|<<|>>|::)"); const regex SPECIAL\_SYMBOL\_PATTERN(R"([{}();,.])");

// Regular expression to identify data types followed by an identifier const regex

DATA\_TYPE\_PATTERN(R"(\b(?:int|float|double|char|long|short|void|bool|unsigned)\s+([A-Za- z\_][A-Za-z0-9\_]\*)\b)");

void remove\_comments(string &content) { size\_t pos;

// Remove single-line comments (//)

while ((pos = content.find("//")) != string::npos) { size\_t end\_pos = content.find("\n", pos); content.erase(pos, end\_pos - pos);

}

// Remove multi-line comments (/\* ... \*/)

while ((pos = content.find("/\*")) != string::npos) { size\_t end\_pos = content.find("\*/", pos);

if (end\_pos == string::npos) break; // If there's no closing \*/, break out. content.erase(pos, end\_pos - pos + 2);

}

}

void analyze\_cpp\_program(const string& file\_path) { ifstream file(file\_path);

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

string content((istreambuf\_iterator<char>(file)), istreambuf\_iterator<char>());

// Remove comments remove\_comments(content);

// Find keywords set<string> keywords;

auto words\_begin = sregex\_iterator(content.begin(), content.end(), KEYWORD\_PATTERN); auto words\_end = sregex\_iterator();

for (sregex\_iterator i = words\_begin; i != words\_end; ++i) { keywords.insert(i->str());

}

// Find identifiers only if preceded by a valid data type set<string> identifiers;

words\_begin = sregex\_iterator(content.begin(), content.end(), DATA\_TYPE\_PATTERN);

for (sregex\_iterator i = words\_begin; i != words\_end; ++i) { identifiers.insert(i->str(1)); // Group 1 contains the identifier after data type

}

// Remove keywords from identifiers (no overlap between keywords and identifiers) for (const auto& keyword : keywords) {

identifiers.erase(keyword);

}

// Find operators vector<string> operators;

words\_begin = sregex\_iterator(content.begin(), content.end(), OPERATOR\_PATTERN);

for (sregex\_iterator i = words\_begin; i != words\_end; ++i) {

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

operators.push\_back(i->str());

}

// Find special symbols vector<string> special\_symbols;

words\_begin = sregex\_iterator(content.begin(), content.end(), SPECIAL\_SYMBOL\_PATTERN);

for (sregex\_iterator i = words\_begin; i != words\_end; ++i) { special\_symbols.push\_back(i->str());

}

// Output the results

cout << "Keywords found: " << keywords.size() << endl; cout << "Keywords: ";

for (const auto& keyword : keywords) cout << keyword << ", "; cout << endl;

cout << "Identifiers found: " << identifiers.size() << endl; cout << "Identifiers: ";

for (const auto& identifier : identifiers) cout << identifier << ", "; cout << endl;

cout << "Operators found: " << operators.size() << endl; cout << "Operators: ";

for (const auto& op : operators) cout << op << ", "; cout << endl;

cout << "Special symbols found: " << special\_symbols.size() << endl; cout << "Special symbols: ";

for (const auto& symbol : special\_symbols) cout << symbol << ", "; cout << endl;

}

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

int main() {

// Example usage:

string file\_path = "sample.c"; // Replace with the path to your C++ program analyze\_cpp\_program(file\_path);

return 0;

}

# Input/Output:

**Keywords found: 5**

**Keywords: float, int, if, main, return Identifiers found: 2**

**Identifiers: a, b Operators: <, >, =, =, >, Special symbols found: 16**

**Special symbols: ., (, ), {, ;, ,, ;, (, ), {, (, ), ;, }, ;, }**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

|  |  |  |
| --- | --- | --- |
| **Program No: 2** | **Roll Number:** 23071A0573 | **Date: 06|08|25** |
| **Program Title:** Lexical Analyzer using lex | | |

# Aim:

To build a lexical analyser using lex

# Description:

**Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools.**

# Program:

**%{**

**#include <stdio.h>**

**%}**

**%%**

**(is|am|are|were|was|be|being|been|to|do|does|did|will|would|should|can|could|has|have|had|go) { printf("Verb: %s\n", yytext); }**

**[a-zA-Z]+ { printf("Non-Verb: %s\n", yytext); }**

**\n {ECHO;}**

**%%**

**int main() {**

**yylex(); // Calls the lexer to start the scanning return 0;**

**}**

**int yywrap() { return 1;**

**}**

**AUTOMATA AND COMPILER DESIGN LABORATORY 2023-27**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

# Input/Output:

**:wq**

**Lex pack.l Cc lex.yy.c**

**./a.out Akshaya**

**Akshaya is not a verb Verb**

**Verb is not a verb Am**

**Am is a verb**

# Program:

%{

#include <stdio.h>

%}

/\* Definitions of token patterns \*/ DIGIT [0-9]

ID [a-zA-Z\_][a-zA-Z0-9\_]\*

KEYWORD (if|else|while) WS [ \t\n]+

%%

{KEYWORD} { printf("Keyword: %s\n", yytext); }

{ID} { printf("Identifier: %s\n", yytext); }

{DIGIT}+ { printf("Number: %s\n", yytext); } "+"|"-"|"\*"|"/" { printf("Operator: %s\n", yytext); }

{WS} { /\* ignore whitespace \*/ }

. { printf("Unknown character: %s\n", yytext); }

%%

int main(int argc, char \*\*argv) { yylex(); // Start lexical analysis return 0;

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

}

int yywrap() { return 1;

}

# Input/Output:

i Identifier: i Am

Identifier: am 2

Number: 2

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

|  |  |  |
| --- | --- | --- |
| **Program No: 3** | **Roll Number:** 23071A0573 | **Date: 20|08|25** |
| **Program Title:** YAAC program | | |

# Aim:

Construct YACC code to implement a calculator

# Description:

**To construct a YAAC code to implement the working and the functioning of a calculator**

# Program:

**Calc.l**

%{

#include <stdio.h> #include <stdlib.h> #include "y.tab.h" extern int yylval;

%}

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; } [ \t]+ { }

\n { return 0; }

. { return yytext[0]; }

%%

int yywrap(void) { return 1; }

# Calc.y

%{

#include <stdio.h> int flag = 0;

int yylex(void);

void yyerror(const char \*s);

%}

%token NUMBER

%left '+' '-'

%left '\*' '/' '%'

%left '(' ')'

## AUTOMATA AND COMPILER DESIGN LABORATORY 2023-27



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

%%

ArithmeticExpression

: E { printf("Result=%d\n", $$); return 0; }

;

E : E '+' E { $$ = $1 + $3; }

| E '-' E { $$ = $1 - $3; }

| E '\*' E { $$ = $1 \* $3; }

| E '/' E { $$ = $1 / $3; }

| E '%' E { $$ = $1 % $3; }

| '(' E ')' { $$ = $2; }

| NUMBER { $$ = $1; }

;

%%

int main(void) {

printf("Enter any arithmetic expression (+,-,\*,/,%% and parentheses):\n"); yyparse();

if (flag == 0) printf("Entered arithmetic expression is Valid\n"); return 0;

}

void yyerror(const char \*s) { (void)s;

printf("Entered arithmetic expression is Invalid\n"); flag = 1;

}

# Input/Output:

bison -d -y calc.y flex calc.l

gcc -o calc y.tab.c lex.yy.c

./calc

Enter any arithmetic expression 12+2

Result = 14

Entered arithmetic expression is valid

./calc

Enter any arithmetic expression 5\*+

Entered arithmetic expression is Invalid



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

|  |  |  |
| --- | --- | --- |
| **Program No: 4** | **Roll Number:** 23071A0573 | **Date: 3|09|25** |
| **Program Title:** First and Follow and Parse table | | |

# Aim:

To Compute First and Follow and construct the parse table

# Description:

**Write a program to compute the First and Follow of a given grammar and construct a parse table for the given grammar**

# Program:

def construct\_parse\_table(prods, firsts, follows): table = defaultdict(dict)

terminals = set() for nt in prods:

for prod in prods[nt]: for symbol in prod:

if symbol not in prods and symbol != 'ε':

terminals.add(symbol) terminals.add('$')

for nt in prods:

for prod in prods[nt]: first\_alpha = set() for symbol in prod:

if symbol not in prods: first\_alpha.add(symbol) break

first\_alpha |= firsts[symbol] - set(['ε']) if 'ε' not in firsts[symbol]:

break

else:

first\_alpha.add('ε')

# Convert production to string for table display prod\_str = "".join(prod)

for t in first\_alpha - set(['ε']):

table[nt][t] = f"{nt}->{prod\_str}"

if 'ε' in first\_alpha:

for t in follows[nt]:

table[nt][t] = f"{nt}->{prod\_str}" return table, terminals



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

# ---------- Print parse table neatly ---------- def print\_parse\_table(table, terminals):

terminals = sorted(list(terminals)) nts = sorted(table.keys())

col\_width = max(10, max(len(t) for t in terminals) + 2)

# Header print("Parse Table:")

print("NT".ljust(10), end='') for t in terminals:

print(t.center(col\_width), end='') print()

print("-" \* (10 + col\_width \* len(terminals)))

for nt in nts: print(nt.ljust(10), end='') for t in terminals:

entry = table[nt].get(t, "") print(entry.center(col\_width), end='')

print()

# Main

n = int(input("Enter number of productions: ")) prods = defaultdict(list)

for \_ in range(n):

lhs, rhs = input().split("->") lhs = lhs.strip()

for r in rhs.split('|'):

tokens = tokenize\_rhs(r.strip()) prods[lhs].append(tokens)

# Eliminate left recursion

prods = eliminate\_left\_recursion(prods) start = list(prods.keys())[0]

# Compute FIRST and FOLLOW firsts = compute\_first(prods)

follows = compute\_follow(prods, firsts, start)

# Construct parse table using transformed productions

parse\_table, terminals = construct\_parse\_table(prods, firsts, follows)

# Display

print("\nGrammar after eliminating left recursion:") for nt in prods:

prods\_str = ["".join(p) for p in prods[nt]] print(f"{nt} -> {' | '.join(prods\_str)}")



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

print("\nFIRST sets:")

for nt in sorted(firsts.keys()):

print(f"FIRST({nt}) = {{ {', '.join(sorted(firsts[nt]))} }}")

print("\nFOLLOW sets:")

for nt in sorted(follows.keys()):

print(f"FOLLOW({nt}) = {{ {', '.join(sorted(follows[nt]))} }}")

print()

print\_parse\_table(parse\_table, terminals)

# Input/Output:

Enter number of productions: 3 E -> E + T | T

T -> T \* F | F

F -> id | (E)

Grammar after eliminating left recursion: E -> TE'

E' -> +TE' | ε

T -> FT'

T' -> \*FT' | ε

F -> id | (E)

FIRST sets: FIRST(E) = { (, id }

FIRST(E') = { +, ε }

FIRST(F) = { (, id }

FIRST(T) = { (, id }

FIRST(T') = { \*, ε }

FOLLOW sets: FOLLOW(E) = { $, ) }

FOLLOW(E') = { $, ) }