**Lexical analyzer c**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

char keywords[][10] = {

"int", "float", "char", "if", "else", "for", "while", "do", "return", "break", "continue", "void"

};

int isKeyword(char \*word) {

for (int i = 0; i < sizeof(keywords)/sizeof(keywords[0]); i++) {

if (strcmp(word, keywords[i]) == 0) {

return 1;

}

}

return 0;

}

int isOperator(char ch) {

return (ch == '+' || ch == '-' || ch == '\*' || ch == '/' ||

ch == '=' || ch == '<' || ch == '>' || ch == '!' || ch == '%');

}

int isSpecialSymbol(char ch) {

return (ch == '(' || ch == ')' || ch == '{' || ch == '}' ||

ch == ';' || ch == ',' || ch == '[' || ch == ']');

}

void lexicalAnalyzer(FILE \*fp) {

char ch;

char buffer[100];

int i = 0;

while ((ch = fgetc(fp)) != EOF) {

// Handling Preprocessor Directives

if (ch == '#') {

printf("Preprocessor Directive\t: ");

printf("#");

while ((ch = fgetc(fp)) != EOF && ch != '\n') {

printf("%c", ch);

}

printf("\n");

continue;

}

// Handling Comments

if (ch == '/') {

char next = fgetc(fp);

if (next == '/') {

printf("Single-line Comment\t: //");

while ((ch = fgetc(fp)) != EOF && ch != '\n') {

printf("%c", ch);

}

printf("\n");

continue;

} else if (next == '\*') {

printf("Multi-line Comment\t: /\*");

while ((ch = fgetc(fp)) != EOF) {

if (ch == '\*' && (next = fgetc(fp)) == '/') {

printf("\*/\n");

break;

}

printf("%c", ch);

}

continue;

} else {

printf("Operator\t\t: /\n");

ungetc(next, fp);

continue;

}

}

// Building Words (Identifiers, Keywords, Numbers)

if (isalnum(ch) || ch == '\_') {

buffer[i++] = ch;

} else {

if (i != 0) {

buffer[i] = '\0';

if (isKeyword(buffer)) {

printf("Keyword\t\t\t: %s\n", buffer);

} else if (isdigit(buffer[0])) {

printf("Number\t\t\t: %s\n", buffer);

} else {

printf("Identifier\t\t: %s\n", buffer);

}

i = 0;

}

// Operators

if (isOperator(ch)) {

char next = fgetc(fp);

if ((ch == '+' && next == '+') || (ch == '-' && next == '-') ||

(ch == '=' && next == '=') || (ch == '!' && next == '=')) {

printf("Operator\t\t: %c%c\n", ch, next);

} else {

printf("Operator\t\t: %c\n", ch);

ungetc(next, fp);

}

}

// Special Symbols

else if (isSpecialSymbol(ch)) {

printf("Special Symbol\t\t: %c\n", ch);

}

// Whitespaces

else if (ch == ' ' || ch == '\n' || ch == '\t') {

// Ignore whitespace

}

// Unknown Characters

else {

if (ch != EOF)

printf("Unknown\t\t\t: %c\n", ch);

}

}

}

}

int main() {

char filename[100];

FILE \*fp;

printf("Enter the filename to analyze: ");

scanf("%s", filename);

fp = fopen(filename, "r");

if (fp == NULL) {

printf("Could not open file %s\n", filename);

return 1;

}

printf("\n--- Lexical Analysis Output ---\n\n");

lexicalAnalyzer(fp);

fclose(fp);

return 0;

}

**Lexical analyzer lex**

%{

#include <stdio.h>

#include <string.h>

void print\_token(const char \*type, const char \*value) {

printf("%-25s : %s\n", type, value);

}

%}

/\* Definitions \*/

KEYWORD int|float|char|if|else|for|while|do|return|break|continue|void

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

NUMBER [0-9]+(\.[0-9]+)?

OPERATOR (\+\+|--|==|!=|<=|>=|=|\+|-|\\*|/|<|>|%)

SPECIAL [\(\){}\[\];,]

%%

"#".\* { print\_token("Preprocessor Directive", yytext); }

"//".\* { print\_token("Single-line Comment", yytext); }

"/\*"([^\*]|\\*+[^\*/])\*\\*+\/ { print\_token("Multi-line Comment", yytext); }

{KEYWORD} { print\_token("Keyword", yytext); }

{IDENTIFIER} { print\_token("Identifier", yytext); }

{NUMBER} { print\_token("Number", yytext); }

{OPERATOR} { print\_token("Operator", yytext); }

{SPECIAL} { print\_token("Special Symbol", yytext); }

[ \t\n]+ { /\* Ignore whitespace \*/ }

. { print\_token("Unknown", yytext); }

%%

int main(int argc, char \*\*argv) {

if (argc > 1) {

FILE \*fp = fopen(argv[1], "r");

if (!fp) {

printf("Could not open file: %s\n", argv[1]);

return 1;

}

yyin = fp;

}

yylex();

return 0;

}

int yywrap() {

return 1;

}

**\*first and follow\***

from collections import defaultdict

EPSILON = 'ε' # Use ε for epsilon (empty string)

END\_MARKER = '$'

class Grammar:

def \_\_init\_\_(self):

self.productions = defaultdict(list)

self.non\_terminals = set()

self.terminals = set()

self.first = defaultdict(set)

self.follow = defaultdict(set)

self.start\_symbol = None

def input\_grammar(self):

print("Enter grammar productions (e.g., E -> T E') one per line. Type 'end' to finish.")

while True:

line = input(">> ").strip()

if line.lower() == 'end':

break

if '->' not in line:

print("Invalid format. Use: NonTerminal -> symbol symbol ...")

continue

lhs, rhs = map(str.strip, line.split("->"))

rhs\_alternatives = rhs.split('|')

for alt in rhs\_alternatives:

symbols = alt.strip().split()

self.productions[lhs].append(symbols)

self.non\_terminals.add(lhs)

for symbol in symbols:

if not symbol.isupper() and symbol != EPSILON:

self.terminals.add(symbol)

self.start\_symbol = next(iter(self.productions))

def compute\_first(self):

for terminal in self.terminals:

self.first[terminal].add(terminal)

for non\_terminal in self.non\_terminals:

self.\_first(non\_terminal)

def \_first(self, symbol):

if symbol in self.first and self.first[symbol]:

return self.first[symbol]

for production in self.productions.get(symbol, []):

for i, sym in enumerate(production):

sym\_first = self.\_first(sym) if sym in self.non\_terminals else {sym}

self.first[symbol].update(sym\_first - {EPSILON})

if EPSILON not in sym\_first:

break

else:

self.first[symbol].add(EPSILON)

return self.first[symbol]

def compute\_follow(self):

self.follow[self.start\_symbol].add(END\_MARKER)

changed = True

while changed:

changed = False

for lhs, rules in self.productions.items():

for rule in rules:

for i, B in enumerate(rule):

if B in self.non\_terminals:

follow\_before = self.follow[B].copy()

rest = rule[i+1:]

if rest:

first\_rest = set()

for sym in rest:

sym\_first = self.first[sym] if sym in self.non\_terminals else {sym}

first\_rest.update(sym\_first - {EPSILON})

if EPSILON not in sym\_first:

break

else:

first\_rest.add(EPSILON)

self.follow[B].update(first\_rest - {EPSILON})

if EPSILON in first\_rest:

self.follow[B].update(self.follow[lhs])

else:

self.follow[B].update(self.follow[lhs])

if follow\_before != self.follow[B]:

changed = True

def display(self):

print("\nFIRST sets:")

for non\_terminal in sorted(self.non\_terminals):

print(f"FIRST({non\_terminal}) = {{ {', '.join(sorted(self.first[non\_terminal]))} }}")

print("\nFOLLOW sets:")

for non\_terminal in sorted(self.non\_terminals):

print(f"FOLLOW({non\_terminal}) = {{ {', '.join(sorted(self.follow[non\_terminal]))} }}")

# Usage

if \_\_name\_\_ == "\_\_main\_\_":

grammar = Grammar()

grammar.input\_grammar()

grammar.compute\_first()

grammar.compute\_follow()

grammar.display()

**LEX TOOL PROGRAMS**

1. %{

#include <stdio.h>

int compound = 0; // Flag to determine if it's a compound sentence

%}

%%

(and|or|but) { compound = 1; } // Match any of the conjunctions and set the flag for compound statement

[ \t\n]+ { /\* Ignore whitespace \*/ } // Ignore spaces, tabs, newlines

. { /\* Ignore other characters \*/ } // Ignore any other character

%%

int main() {

yylex(); // Start lexical analysis

if (compound) {

printf("Compound statement\n"); // If compound is set to 1, print compound statement

} else {

printf("Simple statement\n"); // If no conjunctions are found, print simple statement

}

return 0;

}

**%{**

**#include <stdio.h>**

**long long factorial(int n) {**

**if (n == 0 || n == 1) return 1;**

**return n \* factorial(n - 1);**

**}**

**%}**

**%%**

**[0-9]+ {**

**int num;**

**sscanf(yytext, "%d", &num);**

**printf("Factorial of %d is %lld\n", num, factorial(num));**

**}**

**[ \t\n]+ ; // ignore whitespace**

**. ; // ignore anything else**

**%%**

**int main() {**

**yylex();**

**return 0;**

**}**

**%{**

**#include <stdio.h>**

**int characters = 0, words = 0, lines = 0, blanks = 0;**

**%}**

**%%**

**[ \t]+ { blanks += yyleng; }**

**\n { lines++; }**

**[a-zA-Z0-9]+ { words++; characters += yyleng; }**

**. { characters++; }**

**%%**

**int main() {**

**yylex();**

**printf("Characters: %d\n", characters);**

**printf("Words: %d\n", words);**

**printf("Lines: %d\n", lines);**

**printf("Blanks: %d\n", blanks);**

**return 0;**

**}**

**4) %{**

**#include <stdio.h>**

**int vowels = 0, consonants = 0;**

**%}**

**%%**

**[aeiouAEIOU] { vowels++; }**

**[a-zA-Z] { consonants++; }**

**.|\n { /\* ignore other characters \*/ }**

**%%**

**int main() {**

**yylex();**

**printf("Vowels: %d\n", vowels);**

**printf("Consonants: %d\n", consonants);**

**return 0;**

**}**

**5) %{**

**#include <stdio.h>**

**#include <string.h>**

**%}**

**%%**

**.+ {**

**int len = strlen(yytext);**

**for (int i = len - 1; i >= 0; i--) {**

**putchar(yytext[i]);**

**}**

**putchar('\n');**

**}**

**\n { /\* ignore newlines \*/ }**

**%%**

**int main() {**

**yylex();**

**return 0;**

**}**

**6} %{**

**#include <stdio.h>**

**int count = 0;**

**%}**

**%%**

**\\b(a|an|the)\\b { count++; }**

**[ \t\n]+ { /\* ignore whitespace \*/ }**

**. { /\* ignore other characters \*/ }**

**%%**

**int main() {**

**yylex();**

**printf("Number of articles: %d\n", count);**

**return 0;**

**}**

**\*regular expression c program\***

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <regex.h>**

**int main() {**

**char regex\_pattern[100], input\_string[100];**

**regex\_t regex;**

**int ret;**

**// Get the regular expression and the string from the user**

**printf("Enter the regular expression: ");**

**fgets(regex\_pattern, sizeof(regex\_pattern), stdin);**

**regex\_pattern[strcspn(regex\_pattern, "\n")] = 0; // Remove newline at the end**

**printf("Enter the string: ");**

**fgets(input\_string, sizeof(input\_string), stdin);**

**input\_string[strcspn(input\_string, "\n")] = 0; // Remove newline at the end**

**// Compile the regular expression**

**ret = regcomp(&regex, regex\_pattern, REG\_EXTENDED);**

**if (ret) {**

**printf("Could not compile regex\n");**

**return 1;**

**}**

**// Execute the regular expression match**

**ret = regexec(&regex, input\_string, 0, NULL, 0);**

**if (ret == 0) {**

**printf("Accepted\n");**

**} else if (ret == REG\_NOMATCH) {**

**printf("Rejected\n");**

**} else {**

**char error\_msg[100];**

**regerror(ret, &regex, error\_msg, sizeof(error\_msg));**

**printf("Regex match failed: %s\n", error\_msg);**

**}**

**// Free the memory used by the compiled regex**

**regfree(&regex);**

**return 0;**

**}**

**\*regular expression lex program\***

**%{**

**#include <stdio.h>**

**int match = 0;**

**%}**

**%%**

**^a[0-9]+b$ { match = 1; }**

**. { match = 0; }**

**%%**

**int main() {**

**char input[100];**

**// Get input from the user**

**printf("Enter the string: ");**

**fgets(input, sizeof(input), stdin);**

**input[strcspn(input, "\n")] = 0; // Remove newline at the end**

**// Pass input to Lex**

**YY\_BUFFER\_STATE buffer = yy\_scan\_string(input);**

**yylex(); // Perform lexical analysis**

**// Output the result**

**if (match) {**

**printf("Accepted\n");**

**} else {**

**printf("Rejected\n");**

**}**

**// Free the buffer memory**

**yy\_delete\_buffer(buffer);**

**return 0;**

**}**

**def epsilon\_closure(nfa, start\_states):**

**stack = list(start\_states)**

**closure = set(start\_states)**

**while stack:**

**state = stack.pop()**

**for next\_state in nfa.get(state, {}).get('ε', set()):**

**if next\_state not in closure:**

**closure.add(next\_state)**

**stack.append(next\_state)**

**return closure**

**# Function to read NFA from user**

**def read\_nfa():**

**nfa = {}**

**num\_states = int(input("Enter number of states: "))**

**for \_ in range(num\_states):**

**state = input("Enter state name: ")**

**nfa[state] = {}**

**num\_trans = int(input(f"Enter number of transitions for state {state}: "))**

**for \_ in range(num\_trans):**

**symbol = input(" Enter symbol (use 'ε' for epsilon): ")**

**targets = input(f" Enter target states for symbol '{symbol}' (space-separated): ").split()**

**nfa[state][symbol] = set(targets)**

**return nfa**

**# Main interaction**

**nfa = read\_nfa()**

**start\_states = input("Enter start states (space-separated): ").split()**

**closure = epsilon\_closure(nfa, start\_states)**

**print("Epsilon closure:", closure)**

**output:**

**Enter number of states: 3**

**Enter state name: q0**

**Enter number of transitions for state q0: 1**

**Enter symbol (use 'ε' for epsilon): ε**

**Enter target states for symbol 'ε' (space-separated): q1 q2**

**Enter state name: q1**

**Enter number of transitions for state q1: 1**

**Enter symbol (use 'ε' for epsilon): ε**

**Enter target states for symbol 'ε' (space-separated): q3**

**Enter state name: q2**

**Enter number of transitions for state q2: 0**

**Enter state name: q3**

**Enter number of transitions for state q3: 0**

**Enter start states (space-separated): q0**

**Epsilon closure: {'q0', 'q1', 'q2', 'q3'}**