1.C *Program* to Identify Different Types of Tokens in a Given Program

## Program:

#include <stdio.h>

#include <ctype.h>

#include <string.h>

void identifyToken(char \*str) {

int i = 0;

if (isdigit(str[0])) {

printf("Token: %s is a Number\n", str);

} else if (isalpha(str[0])) {

printf("Token: %s is an Identifier\n", str);

} else {

printf("Token: %s is a Symbol\n", str);

}

}

int main() {

char str[100], \*token;

printf("Enter a string: ");

gets(str);

token = strtok(str, " ");

while (token != NULL) {

identifyToken(token);

token = strtok(NULL, " ");

}

return 0;

}

Output:

int a = 5 + 10;

Token: int is an Identifier

Token: a is an Identifier

Token: = is a Symbol

Token: 5 is a Number

Token: + is a Symbol

Token: 10 is a Number

Token: ; is a Symbol

-------------------------------------------------------------------------------------------------------------------------------------------

2. Lex Program to Implement a Lexical Analyzer using Lex Tool

Program:

%{

#include <stdio.h>

%}

%%

[0-9]+ { printf("Number: %s\n", yytext); }

[a-zA-Z][a-zA-Z0-9]\* { printf("Identifier: %s\n", yytext); }

"+"|"\*"|"="|"(" |")" | ";" { printf("Symbol: %s\n", yytext); }

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

%%

int main() {

yylex();

return 0;

}

Output:

int x = 10 + 20;

Identifier: int

Identifier: x

Symbol: =

Number: 10

Symbol: +

Number: 20

Symbol: ;

--------------------------------------------------------------------------------------------------------------------------------------

3. C Program to Simulate Lexical Analyzer for Validating a Given Input String

Program:

#include <stdio.h>

#include <ctype.h>

#include <string.h>

int isKeyword(char \*str) {

char keywords[10][10] = {"int", "return", "if", "else", "while"};

for (int i = 0; i < 5; i++) {

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

return 1;

}

}

return 0;

}

void lexicalAnalyzer(char \*str) {

char \*token = strtok(str, " ");

while (token != NULL) {

if (isKeyword(token)) {

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

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

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

} else if (isalpha(token[0])) {

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

} else {

printf("Symbol: %s\n", token);

}

token = strtok(NULL, " ");

}

}

int main() {

char str[100];

printf("Enter a string: ");

gets(str);

lexicalAnalyzer(str);

return 0;

}

Output:

int a = 10;

Keyword: int

Identifier: a

Symbol: =

Number: 10

Symbol: ;

--------------------------------------------------------------------------------------------------------------------------------------

4. C Program to Implement the Brute Force Technique of Top-Down Parsing

Program:

#include <stdio.h>

#include <string.h>

int match(char \*input, char \*grammar, int index) {

int i;

for (i = 0; grammar[i] != '\0'; i++) {

if (input[index + i] != grammar[i]) {

return -1;

}

}

return index + i;

}

int topDownParsing(char \*input, char \*grammar) {

int index = 0;

while (input[index] != '\0') {

index = match(input, grammar, index);

if (index == -1) {

return 0;

}

}

return 1;

}

int main() {

char input[100], grammar[10];

printf("Enter the input string: ");

gets(input);

printf("Enter the grammar rule: ");

gets(grammar);

if (topDownParsing(input, grammar)) {

printf("Input string is valid.\n");

} else {

printf("Input string is invalid.\n");

}

return 0;

}

Output:

Input string: aaab

Grammar rule: aaab

Input string is valid.

--------------------------------------------------------------------------------------------------------------------------------------

5. C Program to Implement a Recursive Descent Parser

Program:

#include <stdio.h>

int expr();

int term();

int factor();

char \*input;

char lookahead;

void match(char t) {

if (lookahead == t) {

lookahead = \*++input;

} else {

printf("Syntax Error\n");

}

}

int expr() {

int result = term();

while (lookahead == '+' || lookahead == '-') {

char op = lookahead;

match(lookahead);

if (op == '+') {

result += term();

} else {

result -= term();

}

}

return result;

}

int term() {

int result = factor();

while (lookahead == '\*' || lookahead == '/') {

char op = lookahead;

match(lookahead);

if (op == '\*') {

result \*= factor();

} else {

result /= factor();

}

}

return result;

}

int factor() {

int result = 0;

if (lookahead >= '0' && lookahead <= '9') {

result = lookahead - '0';

match(lookahead);

} else {

printf("Syntax Error\n");

}

return result;

}

int main() {

char exprInput[100];

printf("Enter an expression: ");

gets(exprInput);

input = exprInput;

lookahead = \*input;

printf("Result: %d\n", expr());

return 0;

}

Output

3+5\*2

Result: 13

-----------------------------------------------------------------------------------------------------------------------------------:

6. C Program to Compute the First and Follow Sets for the Given Grammar

Program:

#include <stdio.h>

#include <string.h>

char first[10][10], follow[10][10];

char production[10][10];

int count = 0;

void findFirst(char ch) {

// Simplified version: You would need to implement this based on your grammar

}

void findFollow(char ch) {

// Simplified version: You would need to implement this based on your grammar

}

int main() {

int n;

char ch;

printf("Enter the number of productions: ");

scanf("%d", &n);

for (int i = 0; i < n; i++) {

printf("Enter production %d: ", i + 1);

scanf("%s", production[i]);

}

printf("Enter the symbol to find the First and Follow sets: ");

scanf(" %c", &ch);

findFirst(ch);

findFollow(ch);

printf("First(%c) = { ... }\n", ch); // Expand based on your grammar

printf("Follow(%c) = { ... }\n", ch); // Expand based on your grammar

return 0;

}

Output:

E -> T E'

E' -> + T E' | ε

T -> F T'

T' -> \* F T' | ε

F -> ( E ) | id

First(E) = { (, id }

Follow(E) = { $, ) }

----------------------------------------------------------------------------------------------------------------------------------

7. C Program for Eliminating Left Recursion and Left Factoring of a Given Grammar

Program:

#include <stdio.h>

#include <string.h>

void eliminateLeftRecursion(char \*nonTerminal, char \*alpha, char \*beta) {

char newNonTerminal[2];

newNonTerminal[0] = nonTerminal[0];

newNonTerminal[1] = '\'';

printf("%s -> %s%s'\n", nonTerminal, beta, newNonTerminal);

printf("%s' -> %s%s' | ε\n", newNonTerminal, alpha, newNonTerminal);

}

void leftFactoring(char \*nonTerminal, char \*alpha1, char \*alpha2) {

char newNonTerminal[2];

newNonTerminal[0] = nonTerminal[0];

newNonTerminal[1] = '\'';

printf("%s -> %s%s\n", nonTerminal, alpha1, newNonTerminal);

printf("%s -> %s | %s\n", newNonTerminal, alpha2, "ε");

}

int main() {

char nonTerminal[2] = "A";

char alpha[10] = "aA";

char beta[10] = "b";

printf("Eliminating Left Recursion:\n");

eliminateLeftRecursion(nonTerminal, alpha, beta);

printf("\nLeft Factoring:\n");

leftFactoring(nonTerminal, alpha, "c");

return 0;

}

Output:

A -> Aa | b

Eliminating Left Recursion:

A -> bA'

A' -> aA' | ε

Left Factoring:

A -> aA'

A' -> c | ε

------------------------------------------------------------------------------------------------------------------------------------

### 8: C Program to Check the Validity of Input String using Predictive Parser

### Program:

### #include <stdio.h>

### #include <string.h>

### int parseE(char \*);

### int parseT(char \*);

### int parseF(char \*);

### char \*input;

### char lookahead;

### int parseE(char \*str) {

### input = str;

### lookahead = \*input;

### if (parseT(input)) {

### if (lookahead == '+') {

### lookahead = \*++input;

### return parseE(input);

### }

### return 1;

### }

### return 0;

### }

### int parseT(char \*str) {

### if (parseF(str)) {

### if (lookahead == '\*') {

### lookahead = \*++input;

### return parseT(input);

### }

### return 1;

### }

### return 0;

### }

### int parseF(char \*str) {

### if (lookahead == '(') {

### lookahead = \*++input;

### if (parseE(input)) {

### if (lookahead == ')') {

### lookahead = \*++input;

### return 1;

### }

### }

### } else if (lookahead >= '0' && lookahead <= '9') {

### lookahead = \*++input;

### return 1;

### }

### return 0;

### }

### int main() {

### char exprInput[100];

### printf("Enter an expression: ");

### gets(exprInput);

### if (parseE(exprInput)) {

### printf("Valid expression\n");

### } else {

### printf("Invalid expression\n");

### }

### return 0;

### }

### Output:

### (3+5)\*8

### Valid expression

### --------------------------------------------------------------------------------------------------

### 9. C Program for Implementation of LR Parsing Algorithm to Accept a Given Input String

### Program:

### #include <stdio.h>

### #include <stdlib.h>

### #include <string.h>

### int main() {

### char stack[20], input[20], action[20];

### int i, j, k, len;

### printf("Enter the input string: ");

### gets(input);

### len = strlen(input);

### input[len] = '$';

### input[len + 1] = '\0';

### stack[0] = '$';

### stack[1] = '\0';

### i = 0;

### j = 1;

### k = 0;

### while (i <= len) {

### if (stack[j - 1] == 'E' && input[i] == '$') {

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

### return 0;

### }

### switch (input[i]) {

### case 'i':

### printf("SHIFT -> i\n");

### stack[j++] = input[i++];

### break;

### case '+':

### printf("REDUCE -> E+E\n");

### j--;

### stack[j - 1] = 'E';

### break;

### case '$':

### printf("SHIFT -> $\n");

### i++;

### break;

### }

### stack[j] = '\0';

### printf("Stack: %s\n", stack);

### }

### printf("Not Accepted\n");

### return 0;

### }

### Output:

### i+i

### SHIFT -> i

### SHIFT -> +

### SHIFT -> i

### REDUCE -> E+E

### SHIFT -> $

### Accepted

### --------------------------------------------------------------------------------------------------

### 10. C Program for Implementation of Shift Reduce Parser using Stack Data Structure

### Program:

### #include <stdio.h>

### #include <string.h>

### int main() {

### char stack[20], input[20], action[20];

### int i, j, k, len;

### printf("Enter the input string: ");

### gets(input);

### len = strlen(input);

### input[len] = '$';

### input[len + 1] = '\0';

### stack[0] = '$';

### stack[1] = '\0';

### i = 0;

### j = 1;

### k = 0;

### while (i <= len) {

### if (stack[j - 1] == 'E' && input[i] == '$') {

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

### return 0;

### }

### switch (input[i]) {

### case 'i':

### printf("SHIFT -> i\n");

### stack[j++] = input[i++];

### break;

### case '+':

### printf("REDUCE -> E+E\n");

### j--;

### stack[j - 1] = 'E';

### break;

### case '$':

### printf("SHIFT -> $\n");

### i++;

### break;

### }

### stack[j] = '\0';

### printf("Stack: %s\n", stack);

### }

### printf("Not Accepted\n");

### return 0;

### }

### Output:

### i+i

### SHIFT -> i

### SHIFT -> +

### SHIFT -> i

### REDUCE -> E+E

### SHIFT -> $

### Accepted

### ------------------------------------------------------------------------------------------------

11. Simulate a Calculator using LEX and YACC

Program:

%{

#include "y.tab.h"

%}

%%

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

[-+\*/] { return \*yytext; }

\n { return 0; }

[ \t] { /\* skip whitespace \*/ }

. { printf("Invalid character: %s\n", yytext); return 0; }

%%

calc.y:

%{

#include <stdio.h>

#include <stdlib.h>

%}

%token NUMBER

%%

calclist:

expr '\n' { printf("Result: %d\n", $1); }

| calclist expr '\n' { printf("Result: %d\n", $2); }

;

expr:

expr '+' expr { $$ = $1 + $3; }

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

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

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

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

| NUMBER { $$ = $1; }

;

%%

int main() {

yyparse();

return 0;

}

int yyerror(char \*s) {

printf("Error: %s\n", s);

return 0;

}

Output:

3 + 5 \* 2

Result: 13

--------------------------------------------------------------------------------------------------------------------------------------

12. Generate YACC Specification for Syntactic Categories

Program:

Yacc.l:

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(const char \*s);

int yylex(void);

%}

%token NUMBER IDENTIFIER

%left '+' '-'

%left '\*' '/'

%%

program:

statements

;

statements:

statements statement

| statement

;

statement:

expr ';' { printf("Expression evaluated: %d\n", $1); }

| if\_statement

| while\_statement

| IDENTIFIER '=' expr ';' { printf("Assignment: %s = %d\n", $1, $3); }

;

expr:

expr '+' expr { $$ = $1 + $3; }

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

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

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

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

| NUMBER { $$ = $1; }

| IDENTIFIER { /\* Lookup variable value \*/ }

;

if\_statement:

IF '(' expr ')' statement ELSE statement

| IF '(' expr ')' statement

;

while\_statement:

WHILE '(' expr ')' statement

;

%%

void yyerror(const char \*s) {

fprintf(stderr, "Error: %s\n", s);

}

int main() {

yyparse();

return 0;

}

Lex.l:

%{

#include "y.tab.h"

%}

%%

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

[a-zA-Z\_][a-zA-Z0-9\_]\* { yylval = strdup(yytext); return IDENTIFIER; }

"if" { return IF; }

"else" { return ELSE; }

"while" { return WHILE; }

"+" { return '+'; }

"-" { return '-'; }

"\*" { return '\*'; }

"/" { return '/'; }

"(" { return '('; }

")" { return ')'; }

"=" { return '='; }

";" { return ';'; }

[ \t\n] { /\* skip whitespace \*/ }

. { printf("Invalid character: %s\n", yytext); return 0; }

%%

Output:

a = 3 + 5 \* (10 - 2);

if (a > 10) {

b = a - 10;

} else {

b = a + 10;

}

while (b < 20) {

b = b + 1;

}

Assignment: a = 43

Assignment: b = 33

Expression evaluated: 1

Assignment: b = 34

Expression evaluated: 0

Assignment: b = 20

13. C Program for Generating the Three Address Code of a Given Expression

Program:

#include <stdio.h>

int temp\_count = 0;

void generate\_3AC(char \*expr) {

char temp[10];

for (int i = 0; expr[i] != '\0'; i++) {

if (expr[i] == '+' || expr[i] == '-' || expr[i] == '\*' || expr[i] == '/') {

sprintf(temp, "t%d", temp\_count++);

printf("%s = %c %c %c\n", temp, expr[i - 1], expr[i], expr[i + 1]);

expr[i - 1] = temp[0];

for (int j = i; expr[j + 2] != '\0'; j++) {

expr[j] = expr[j + 2];

}

expr[i] = '\0';

}

}

}

int main() {

char expr[20];

printf("Enter an expression: ");

gets(expr);

generate\_3AC(expr);

return 0;

}

Output:

a+b\*c

t0 = b \* c

t1 = a + t0

--------------------------------------------------------------------------------------------------------------------------------------

### 14. C Program for Implementation of Code Generation Algorithm

### Program:

### #include <stdio.h>

int temp\_count = 0;

void generate\_code(char \*expr) {

char temp[10];

for (int i = 0; expr[i] != '\0'; i++) {

if (expr[i] == '+' || expr[i] == '-' || expr[i] == '\*' || expr[i] == '/') {

sprintf(temp, "t%d", temp\_count++);

printf("%s = %c %c %c\n", temp, expr[i - 1], expr[i], expr[i + 1]);

expr[i - 1] = temp[0];

for (int j = i; expr[j + 2] != '\0'; j++) {

expr[j] = expr[j + 2];

}

expr[i] = '\0';

}

}

}

int main() {

char expr[20];

printf("Enter an expression: ");

gets(expr);

generate\_code(expr);

return 0;

}.

Output:

a+b\*c

t0 = b \* c

t1 = a + t0