LEXICAL ANALYSER IN C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <ctype.h>
// Function prototypes
bool isDelimiter(char ch);
bool isOperator(char ch);
bool isValidIdentifier(char* str);
bool isKeyword(char* str);
bool isInteger(char* str);
bool isRealNumber(char* str);
char* substring(char* str, int left, int right);
void parse(char* str);

// Main function to read from file and parse its contents
int main() {
    FILE *file = fopen("input.txt", "r");  // Open input.txt in read mode
    if (file == NULL) {
        printf("Error: Could not open input.txt\n");
        return 1;
    }
    char str[1000];  // Buffer to store the contents of the file

    printf("Reading from input.txt:\n");
    while (fgets(str, sizeof(str), file) != NULL) {  // Read the file line by line
        printf("%s", str);  // Display the contents (optional)
        parse(str);  // Analyze each line
    }

    fclose(file);  // Close the file
    return 0;
}

// Check if the character is a delimiter
bool isDelimiter(char ch) {
    return (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
            ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
            ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
            ch == '[' || ch == ']' || ch == '{' || ch == '}');
}

// Check if the character is an operator (including multi-character)
bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' ||
            ch == '>' || ch == '<' || ch == '=' || ch == '%');
}
```

```c
// Check if the string is a valid identifier
bool isValidIdentifier(char* str) {
    if (!isalpha(str[0]) && str[0] != '_') {
        return false;
    }
    for (int i = 1; i < strlen(str); i++) {
        if (!isalnum(str[i]) && str[i] != '_') {
            return false;
        }
    }
    return true;
}

// Check if the string is a keyword
bool isKeyword(char* str) {
    const char* keywords[] = {
        "if", "else", "while", "do", "break", "continue", "int", "double",
        "float", "return", "char", "case", "sizeof", "long", "short",
        "typedef", "switch", "unsigned", "void", "static", "struct", "goto"
    };
    for (int i = 0; i < 22; ++i) {
        if (strcmp(str, keywords[i]) == 0) {
            return true;
        }
    }
    return false;
}

// Check if the string is an integer
bool isInteger(char* str) {
    int len = strlen(str);
    if (len == 0) return false;

    for (int i = 0; i < len; i++) {
        if (!isdigit(str[i])) return false;
    }
    return true;
}

// Check if the string is a real number
bool isRealNumber(char* str) {
    int len = strlen(str);
    bool hasDecimal = false;

    for (int i = 0; i < len; i++) {
        if (str[i] == '.') {
```

```
 if (hasDecimal) return false;  // More than one decimal
point
            hasDecimal = true;
        } else if (!isdigit(str[i])) {
            return false;
        }
    }
    return hasDecimal;
}

// Extract a substring from the input string
char* substring(char* str, int left, int right) {
    char* subStr = (char*)malloc((right - left + 2) *
sizeof(char));
    for (int i = left; i <= right; i++) {
        subStr[i - left] = str[i];
    }
    subStr[right - left + 1] = '\0';
    return subStr;
}

// Parse the input string and identify tokens
void parse(char* str) {
    int left = 0, right = 0;
    int len = strlen(str);

    while (right <= len) {
        // Skip whitespace
        while (right < len && isspace(str[right])) {
            right++;
        }
        left = right;

        // If the line starts with a comment, skip it
        if (str[left] == '/' && str[left + 1] == '/') {
            printf("// this is a comment (ignored)\n");
            break;  // Stop processing this line
        }

        // Move `right` to find the end of the current token
        while (right < len && !isDelimiter(str[right])) {
            right++;
        }

        // Extract the current token
        if (left < right) {
            char* subStr = substring(str, left, right - 1);

            // Strip trailing whitespace from the token
            size_t tokenLen = strlen(subStr);
                        if (tokenLen > 0 &&
isspace(subStr[tokenLen - 1])) {
                subStr[tokenLen - 1] = '\0';
            }

            if (isKeyword(subStr)) {
                printf("'%s' is a keyword\n", subStr);
            } else if (isInteger(subStr)) {
                 printf("'%s' is an integer\n", subStr);
            } else if (isRealNumber(subStr)) {
                printf("'%s' is a real number\n", subStr);
            } else if (isValidIdentifier(subStr)) {
                printf("'%s' is a valid identifier\n", subStr);
            } else {
                printf("'%s' is not a valid identifier\n", subStr);
            }

            free(subStr);
        }

        // If the delimiter is an operator, print it
        if (isOperator(str[right])) {
            printf("'%c' is an operator\n", str[right]);
        }

        // Move to the next token
        right++;
    }
}
```

```
// input.txt
'int' is a keyword
'a' is a valid identifier
'=' is an operator
'5' is an integer
// this is a comment (ignored)
```

```
gcc main.c
./a.out
```

```
'int' is a keyword
'a' is a valid identifier
'=' is an operator
'5' is an integer
// this is a comment (ignored)
```

LEXICAL ANALYSER USING LEX

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Counter variables
int chars = 0;
int words = 0;
int lines = 0;
int spaces = 0;
int special_chars = 0;
int printlnCount = 0;

// Function to update counters
void count() {
    chars += yyleng;
}
%}

%%
\n              { lines++; chars++; }
[a-zA-Z]+       { words++; count(); printf("Word: %s\n", yytext); }
" "             { spaces++; chars++; }
[,\.\?!;:]      { special_chars++; count(); printf("Special Character: %s\n", yytext); }
\"[^\"]*\"      {
                    count();
                    printf("Quoted String: %s\n", yytext);
                    // Check for "println" in quoted strings
                    if(strstr(yytext, "println") != NULL) {
                        printlnCount++;
                    }
                }
println         { printlnCount++; printf("Found 'println'\n"); }
.               { count(); }

%%

int main(int argc, char **argv) {
    if(argc != 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    FILE *file = fopen(argv[1], "r");
    if(!file) {
        printf("Could not open file: %s\n", argv[1]);
        return 1;
    }

    yyin = file;

    printf("\nBeginning Analysis of file: %s\n", argv[1]);
    printf("----------------------------------------\n\n");

    yylex();

    printf("\n----------------------------------------\n");
    printf("Analysis Results:\n");
    printf("----------------------------------------\n");
    printf("Total Characters: %d\n", chars);
    printf("Total Words: %d\n", words);
    printf("Total Lines: %d\n", lines + 1); // +1 for the last line if it doesn't end with \n
    printf("Total Spaces: %d\n", spaces);
    printf("Total Special Characters: %d\n", special_chars);
    printf("Occurrences of 'println': %d\n", printlnCount);

    fclose(file);
    return 0;
}

int yywrap() {
    return 1;
}
```

```
// input.txt
Hello, this is a sample input file. It contains multiple lines and words.
Let's see how the lexical analyzer works.
"Here is a line with println in it." Another Line without the specific substring.
```

```
lex main.l
gcc lex.yy.c -o a.out
./a.out < input.txt
```

LOOPS

```
// LEX FILE - parser.l
%{
#include "y.tab.h"  // Include the header file generated by YACC
%}

%option noyywrap

%%
[ \t\n]+              ; // Ignore whitespace
"while"               { return WHILE; }
"<="                  { return LE; }
">="                  { return GE; }
"=="                  { return EQ; }
"!="                  { return NE; }
"&&"                  { return AND; }
"||"                  { return OR; }
"("                   { return '('; }
")"                   { return ')'; }
"{"                   { return '{'; }
"}"                   { return '}'; }
";"                   { return ';'; }
[0-9]+                { yylval = atoi(yytext); return NUM; } // Return numbers
[a-zA-Z_][a-zA-Z0-9_]* { yylval = strdup(yytext); return ID; } // Return identifiers
"<"                   { return '<'; }
">"                   { return '>'; }
"+"                   { return '+'; }
"-"                   { return '-'; }
"*"                   { return '*'; }
"/"                   { return '/'; }
.                     { return yytext[0]; } // Return any other single character
%%

// Error handling for invalid characters
int yyerror(char* msg) {
    fprintf(stderr, "Error: %s\n", msg);
    return 0;
}
```

```
//YACC FILE - parser.y
%{
#include <stdio.h>
#include <stdlib.h>

int yylex(void);
void yyerror(char* msg);
%}

%token ID NUM WHILE LE GE EQ NE OR AND

%left '+' '-'
%left '*' '/'
%right UMINUS
%nonassoc '<' '>' LE GE EQ NE
%nonassoc AND OR

%%
S: ST { printf("Input accepted\n"); exit(0); }
  ;

ST: ST ';'
   | WHILE '(' E ')' '{' ST '}'
   | ID '=' E ';'
   ;

E: E '+' E
 | E '-' E
 | E '*' E
 | E '/' E
 | '-' E %prec UMINUS
 | ID
 | NUM
 | '(' E ')'
 | E '<' E
 | E '>' E
 | E LE E
 | E GE E
 | E EQ E
 | E NE E
 | E OR E
 | E AND E
 ;

%%

int main() {
    printf("Enter the expression: ");
    yyparse();
    return 0;
}

void yyerror(char* msg) {
    fprintf(stderr, "Invalid input: %s\n", msg);
}
```

```
yacc -d parser.y
lex lexer.l
gcc y.tab.c lex.yy.c -o parser -ll
./parser
```

```
Enter the Expression : while(I<=5){a=a+5;}
INPUT ACCEPTED
```

CALCULATOR

```
// YACC FILE - arith.y

%{
#include <stdio.h>
#include <stdlib.h>

void yyerror(const char *s);
int yylex(void);
%}

%token NUMBER
%token PLUS MINUS TIMES DIVIDE LPAREN
RPAREN

%left PLUS MINUS
%left TIMES DIVIDE
%right UMINUS

%%
input:
    /* Empty */
    | input line
;

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

expr:
    NUMBER
    | expr PLUS expr { $$ = $1 + $3; }
    | expr MINUS expr { $$ = $1 - $3; }
    | expr TIMES expr { $$ = $1 * $3; }
    | expr DIVIDE expr { $$ = $1 / $3; }
    | MINUS expr %prec UMINUS { $$ = -$2; }
    | LPAREN expr RPAREN { $$ = $2; }
;

%%

int main(void) {
    yyparse();
    return 0;
}

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

```
// LEX FILE - lex.l

%{
#include "y.tab.h"
#include <stdlib.h>
%}

%%
[0-9]+    { yylval = atoi(yytext); return NUMBER; }
"+"      { return PLUS; }
"-"      { return MINUS; }
"*"      { return TIMES; }
"/"      { return DIVIDE; }
"("      { return LPAREN; }
")"      { return RPAREN; }
[ \t\n]  { /* skip whitespace */ }
.        { /* ignore unrecognized characters */ }

%%

int yywrap(void) {
    return 1;
}
```

```
// input.txt

3+5*(2-8)
10/2+3
```

```
yacc -d arith.y
lex lex.l
gcc -o arith y.tab.c lex.yy.c -lfl
./arith
```

```
Result = -27
Result = 8
```

8086 BACKEND ASSEMBLY

```
#include <stdio.h>
#include <string.h>

int main() {
   char icode[10][30], str[20], opr[10];
   int i = 0;

   printf("\nEnter the set of intermediate code (terminated
by 'exit'):\n");

   // Reading the input until 'exit' is encountered
   do {
      scanf("%s", icode[i]);
   } while (strcmp(icode[i++], "exit") != 0);

   printf("Target code generation\n");

   // Process each instruction
   for (int j = 0; j < i - 1; j++) {  // i - 1 to skip 'exit'
      strcpy(str, icode[j]);
      switch (str[3]) {
         case '+':
            strcpy(opr, "ADD");
            break;
         case '-':
            strcpy(opr, "SUB");
            break;
         case '*':
            strcpy(opr, "MUL");
            break;
         case '/':
            strcpy(opr, "DIV");
            break;
         default:
            strcpy(opr, "UNKNOWN");
            break;
      }

      // Print the assembly code based on the
intermediate code format
      if (strcmp(opr, "UNKNOWN") != 0) {
         printf("\n\tMOV %c, R%d", str[0], 0); // Assuming
str[0] is the destination
         printf("\n\t%s %c, R%d", opr, str[2], 1); // str[2] is
the source operand
         printf("\n\tMOV R%d, %c", 1, str[4]); // Assuming
str[4] is the destination
      }
   }

   return 0;
}
```

```
gcc main.c
./a.out

Enter the set of intermediate code (terminated by 'exit'):
a=b+c
d=e-f
exit


Target code generation

      MOV a, R0
      ADD b, R1
      MOV R1, c
      MOV d, R0
      SUB e, R1
      MOV R1, f
```

RECURSIVE DESCENT

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void E();        // Forward declaration for E
void Eprime();   // Forward declaration for E'
void T();        // Forward declaration for T
void Tprime();   // Forward declaration for T'
void check();    // Function to check if character is
alphanumeric or '('

char expression[10];
int count, flag;

int main() {
    count = 0;
    flag = 0;

    printf("\nEnter an Algebraic Expression: ");
    scanf("%s", expression);  // Read expression as string

    E();  // Start with the non-terminal 'E'

    if ((strlen(expression) == count) && (flag == 0)) {
        printf("\nThe Expression %s is Valid\n", expression);
    } else {
        printf("\nThe Expression %s is Invalid\n",
expression);
    }

    return 0;
}

void E() {
    T();        // E -> T E'
    Eprime();   // Continue with E'
}

void Eprime() {
    if (expression[count] == '+') {
        count++;     // Consume '+'
        T();         // Parse T
        Eprime();    // Parse E' recursively
    }
}

void T() {
    check();    // T -> check T'
    Tprime();   // Continue with T'
}
```

```
void Tprime() {
    if (expression[count] == '*') {
        count++;     // Consume '*'
        check();     // Parse factor (alphanumeric or
parenthesis)
        Tprime();    // Parse T' recursively
    }
}

void check() {
    if (isalnum(expression[count])) {
        count++;     // Consume alphanumeric character
    } else if (expression[count] == '(') {
        count++;     // Consume '('
        E();         // Parse sub-expression
        if (expression[count] == ')') {
            count++; // Consume ')'
        } else {
            flag = 1; // Set flag for invalid expression
        }
    } else {
        flag = 1;     // Set flag for invalid expression
    }
}
```

SHIFT REDUCE PARSER

```
#include <stdio.h>
#include <string.h>

int k = 0, z = 0, i = 0, j = 0;
char a[16], ac[20], stk[15], act[10];

void check();

int main() {
    printf("GRAMMAR: E -> E+E | E*E | (E) | id\n");
    printf("Enter input string: ");
    fgets(a, sizeof(a), stdin);  // Use fgets for safer input
    a[strcspn(a, "\n")] = 0;     // Remove newline character

    strcpy(act, "SHIFT");
    printf("Stack\tInput\tAction\n");

    for (k = 0, i = 0, j = 0; j < strlen(a); k++, i++, j++) {
        if (a[j] == 'i' && a[j + 1] == 'd') {
            stk[i] = a[j];
            stk[i + 1] = a[j + 1];
            stk[i + 2] = '\0';
            a[j] = ' ';
            a[j + 1] = ' ';
            printf("\n$%s\t%s$\t%s id", stk, a, act);
            check();
        } else {
            stk[i] = a[j];
            stk[i + 1] = '\0';
            a[j] = ' ';
            printf("\n$%s\t%s$\t%s symbols", stk, a, act);
            check();
        }
    }
    check();  // Final check after the loop for any remaining
reductions
    return 0;
}

void check() {
    strcpy(ac, "REDUCE TO E");
    for (z = 0; z < strlen(stk); z++) {
        if (stk[z] == 'i' && stk[z + 1] == 'd') {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            i--;
        }
    }
 for (z = 0; z < strlen(stk); z++) {
        if (stk[z] == 'E' && stk[z + 1] == '+' && stk[z + 2] ==
'E') {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            i -= 2;
        }
    }

    for (z = 0; z < strlen(stk); z++) {
        if (stk[z] == 'E' && stk[z + 1] == '*' && stk[z + 2] ==
'E') {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            i -= 2;
        }
    }

    for (z = 0; z < strlen(stk); z++) {
        if (stk[z] == '(' && stk[z + 1] == 'E' && stk[z + 2] ==
')') {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            i -= 2;
        }
    }
}
```