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

#define MAX_LENGTH 100

// this function check for a delimiter(it is a piece of data
// that seprated it from other) to peform some specif case
// on it
bool isDelimiter(char chr)
{
   return (chr == ' ' || chr == '+' || chr == '-'
          || chr == '*' || chr == '/' || chr == ','
          || chr == ';' || chr == '%' || chr == '>'
          || chr == '<' || chr == '=' || chr == '('
          || chr == ')' || chr == '[' || chr == ']'
          || chr == '{' || chr == '}');
}

// this function check for a valid identifier eg:- +,-* etc
bool isOperator(char chr)
{
   return (chr == '+' || chr == '-' || chr == '*'
          || chr == '/' || chr == '>' || chr == '<'
          || chr == '=');
}

// this function check for an valid identifier
bool isValidIdentifier(char* str)
```

```
{
    return (str[0] != '0' && str[0] != '1' && str[0] != '2'
            && str[0] != '3' && str[0] != '4'
            && str[0] != '5' && str[0] != '6'
            && str[0] != '7' && str[0] != '8'
            && str[0] != '9' && !isDelimiter(str[0]));
}

// 32 Keywords are checked in this function and return the
// result accordingly
bool isKeyword(char* str)
{
    const char* 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" };
    for (int i = 0;
         i < sizeof(keywords) / sizeof(keywords[0]); i++) {
        if (strcmp(str, keywords[i]) == 0) {
            return true;
        }
    }
    return false;
}

// check for an integer value
```

```
bool isInteger(char* str)
{
    if (str == NULL || *str == '\0') {
        return false;
    }
    int i = 0;
    while (isdigit(str[i])) {
        i++;
    }
    return str[i] == '\0';
}

// trims a substring from a given string's start and end
// position
char* getSubstring(char* str, int start, int end)
{
    int length = strlen(str);
    int subLength = end - start + 1;
    char* subStr
        = (char*)malloc((subLength + 1) * sizeof(char));
    strncpy(subStr, str + start, subLength);
    subStr[subLength] = '\0';
    return subStr;
}

// this function parse the input
int lexicalAnalyzer(char* input)
{
    int left = 0, right = 0;
    int len = strlen(input);
```

```
while (right <= len && left <= right) {
    if (!isDelimiter(input[right]))
        right++;

    if (isDelimiter(input[right]) && left == right) {
        if (isOperator(input[right]))
            printf("Token: Operator, Value: %c\n",
                   input[right]);

        right++;
        left = right;
    }
    else if (isDelimiter(input[right]) && left != right
             || (right == len && left != right)) {
        char* subStr
            = getSubstring(input, left, right - 1);

        if (isKeyword(subStr))
            printf("Token: Keyword, Value: %s\n",
                   subStr);

        else if (isInteger(subStr))
            printf("Token: Integer, Value: %s\n",
                   subStr);

        else if (isValidIdentifier(subStr)
                 && !isDelimiter(input[right - 1]))
            printf("Token: Identifier, Value: %s\n",
                   subStr);

        else if (!isValidIdentifier(subStr)
                  && !isDelimiter(input[right - 1]))
              printf("Token: Unidentified, Value: %s\n",
                    subStr);
          left = right;
      }
   }
   return 0;
}

// main function
int main()
{
   // Input 01
   char lex_input[MAX_LENGTH] = "int a = b + c";
   printf("For Expression \"%s\":\n", lex_input);
   lexicalAnalyzer(lex_input);
   printf(" \n");
   // Input 02
   char lex_input01[MAX_LENGTH]
      = "int x=ab+bc+30+switch+ 0y ";
   printf("For Expression \"%s\":\n", lex_input01);
   lexicalAnalyzer(lex_input01);
   return (0);
}
```

Output :

```
For Expression 1:
Token: Keyword, Value: int
Token: Identifier, Value: a
Token: Operator, Value: =
Token: Identifier, Value: b
Token: Operator, Value: +
Token: Identifier, Value: c
```

For Expression 2:

```
Token: Keyword, Value: int
Token: Identifier, Value: x
Token: Operator, Value: =
Token: Identifier, Value: ab
Token: Operator, Value: +
Token: Identifier, Value: bc
Token: Operator, Value: +
Token: Integer, Value: 30
Token: Operator, Value: +
Token: Keyword, Value: switch
Token: Operator, Value: +
Token: Unidentified, Value: 0y
```