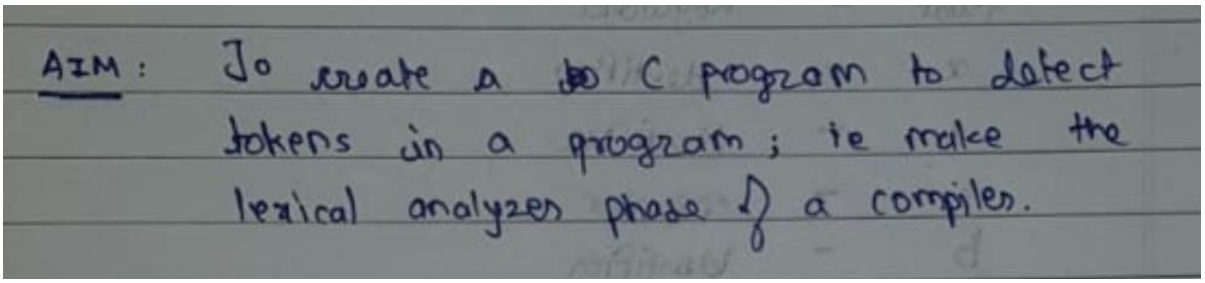
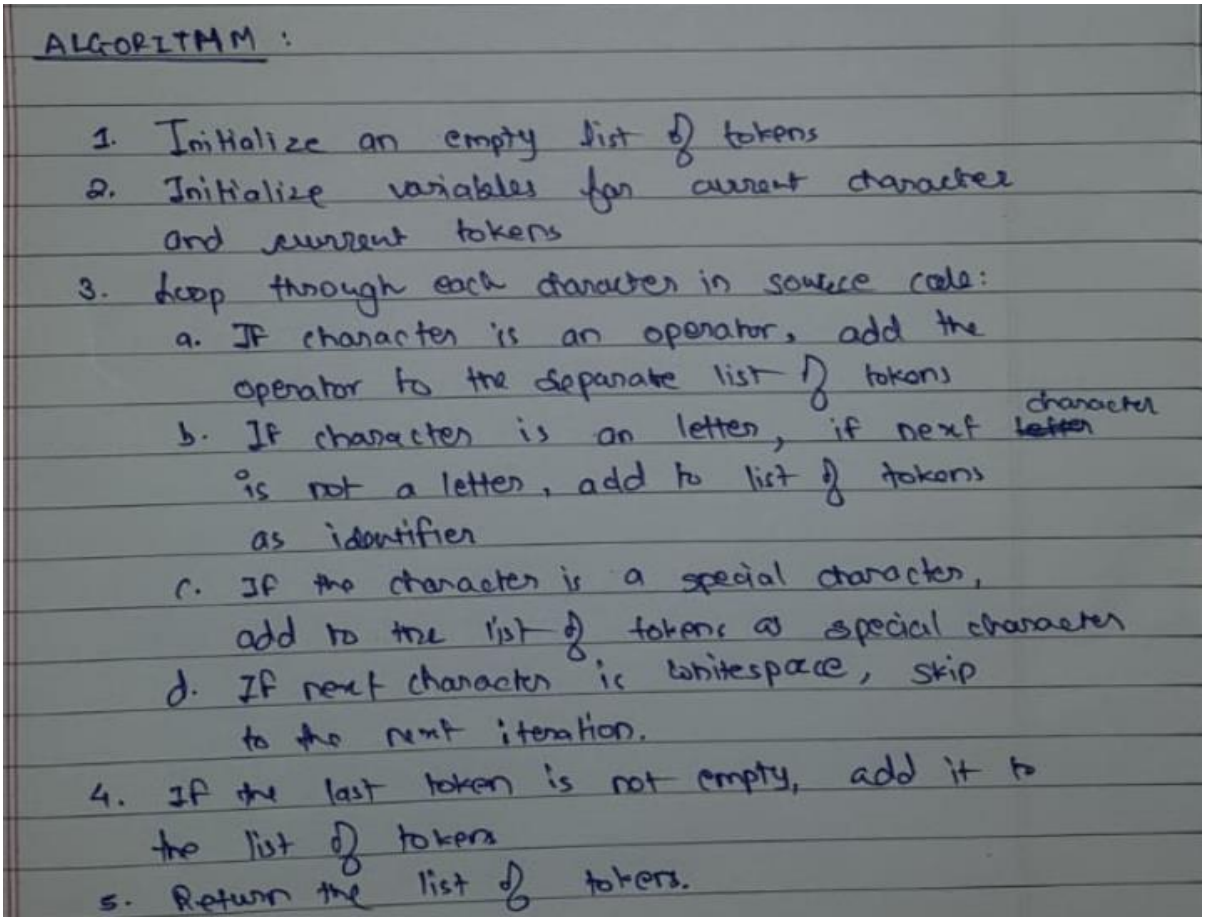
Lexical Analysis using C
Compiler Design – Lab 1

Name: Ojas Patil
Reg No: 21BAI1106

AIM

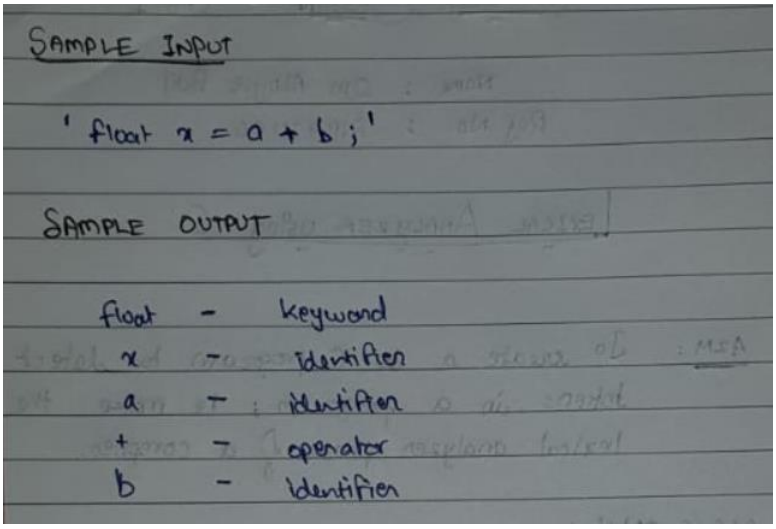

AIM: To create a C program to detect tokens in a program; ie make the lexical analyzer phase of a compiler.

ALGORITHM


ALGORITHM:

1. Initialize an empty list of tokens
2. Initialize variables for current character and current tokens
3. Loop through each character in source code:
   a. If character is an operator, add the operator to the separate list of tokens
   b. If character is an letter, if next character is not a letter, add to list of tokens as identifier
   c. If the character is a special character, add to the list of tokens as special character
   d. If next character is whitespace, skip to the next iteration.
4. If the last token is not empty, add it to the list of tokens
5. Return the list of tokens.

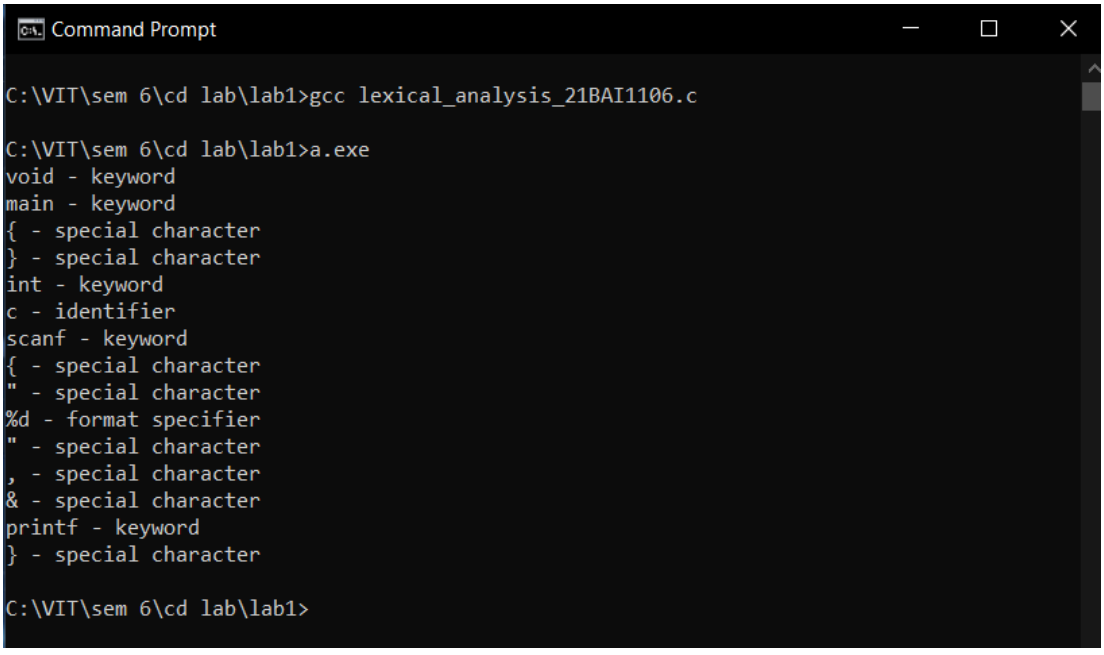
Sample Input and Output



Input

```
char input[] = "void main()\n"
               "{\n"
               "    int c;\n"
               "    scanf(\"%d\", &c);\n"
               "    printf(\"%d\", c);\n"
               "}\n";
```

Output Screenshot



Program Explanation

This C program is a simple lexical analyzer that processes a given input source code and categorizes the lexemes into keywords, identifiers, special characters, and format specifiers. Here's a 5-line explanation:

1. **Tokenization**: The program uses strtok to tokenize the input source code based on specified delimiters (spaces, tabs, newlines, parentheses, quotes, commas, and braces).
2. **Keyword Identification**: It checks if each token matches predefined keywords ("void," "main," "int," "scanf," "printf"), and if so, it classifies it as a keyword.
3. **Special Character Detection**: It identifies special characters such as parentheses, quotes, commas, and braces and categorizes them accordingly.
4. **Format Specifier Handling**: For tokens starting with '&' or '%', it recognizes them as special characters and skips the next token (which is part of the format specifier), printing it as a format specifier.
5. **Output Printing**: The program then prints each token along with its corresponding category, such as "keyword," "identifier," "special character," or "format specifier."

Source Code

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

int isKeyword(char *lexeme)
{
    char keywords[][10] = {"void", "main", "int", "scanf", "printf"};
    int i, flag = 0;

    for (i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++)
    {
        if (strcmp(lexeme, keywords[i]) == 0)
        {
            flag = 1;
            break;
        }
    }
    return flag;
}

int main()
{
    char input[] = "void main()\n"
                   "{\n"
```

```
                    "    int c;\n"
                    "    scanf(\"%d\", &c);\n"
                    "    printf(\"%d\", c);\n"
                    "}\n";
    char *token = strtok(input, " \t\n()\";,{}");

    while (token != NULL)
    {
        if (strcmp(token, "void") == 0 || strcmp(token, "main") == 0 ||
strcmp(token, "int") == 0 || strcmp(token, "scanf") == 0 || strcmp(token,
"printf") == 0)
        {
            printf("%s - keyword\n", token);
        }
        else if (token[0] == '(' || token[0] == ')' || token[0] == '"' ||
token[0] == ',' || token[0] == '{' || token[0] == '}')
        {
            printf("%s - special character\n", token);
        }
        else if (token[0] == '&' || token[0] == '%')
        {
            printf("%s - special character\n", token);
            token = strtok(NULL, " \t\n()\";,{}");
            printf("%s - format specifier\n", token);
        }
        else
        {
            printf("%s - identifier\n", token);
        }
        token = strtok(NULL, " \t\n()\";,{}");
    }
    return 0;
}
```

Conclusion

Thus, we have studied and created a C program which functions as a basic lexical analyzer, categorizing tokens from input source code into keywords, identifiers, special characters, and format specifiers, providing a simple representation of lexical elements in the code.