**EXPERIMENT-4**

**Program:**

Write a c program to identifiers, keywords, white spaces and comments (Lexical analysis) in a program.

**Theory:**

To remove identifiers, keywords, white spaces, and comments during lexical analysis, you can follow these general steps:

1. **Define Token Types:**
   * Identify and define the different types of tokens in your programming language, such as keywords, identifiers, operators, literals, and comments.
2. **Regular Expressions:**
   * Use regular expressions to define patterns for each type of token. For example, you might use a regular expression to match keywords, identifiers, or comments.
3. **Lexical Rules:**
   * Establish lexical rules that specify how to recognize and categorize tokens based on the defined patterns. These rules serve as the foundation for the lexical analyzer.
4. **Tokenization:**
   * Implement a lexical analyzer that scans the source code and recognizes tokens based on the lexical rules. This process involves identifying keywords, identifiers, operators, literals, and comments.
5. **Filtering Out Unwanted Tokens:**
   * Once you've identified the different types of tokens, filter out the ones you want to remove. This may include discarding identifiers, keywords, white spaces, and comments.
6. **Output:**
   * Provide the filtered list of tokens as output, excluding the specified elements. This processed list can then be passed on to the next stage of the compiler for further analysis and processing.
7. **Handling White Spaces:**
   * If white spaces need to be removed, simply skip over them during the tokenization process or filter them out in the token list.
8. **Handling Comments:**
   * If comments need to be removed, you can choose to ignore or skip them during the tokenization process. Regular expressions can help in recognizing and ignoring comment blocks.

Remember that the exact implementation details may vary based on the programming language and the tools you are using for lexical analysis. Many compiler construction tools, such as Lex and Flex, provide convenient ways to define token patterns and rules for lexical analysis.

**Pseudo code:**

procedure tokenize(source\_code):

initialize an empty list for tokens

current\_position = 0

while current\_position < length(source\_code):

token, new\_position = match\_token(source\_code, current\_position)

if token is not null:

tokens.append(token)

current\_position = new\_position

filtered\_tokens = filter\_unwanted\_tokens(tokens)

return filtered\_tokens

procedure match\_token(source\_code, current\_position):

keyword\_pattern = /\b(if|else|while|for)\b/

identifier\_pattern = /[a-zA-Z\_][a-zA-Z0-9\_]\*/

comment\_pattern = /(\/\/.\*|\/\\*[\s\S]\*?\\*\/)/

master\_pattern = `${keyword\_pattern}|${identifier\_pattern}|${comment\_pattern}`

match\_result = match(master\_pattern, source\_code, current\_position)

if match\_result is not null:

matched\_token = match\_result.group()

new\_position = current\_position + length(matched\_token)

return matched\_token, new\_position

// No match found

return null, current\_position + 1

procedure filter\_unwanted\_tokens(tokens):

// Filter out unwanted tokens (identifiers, keywords, comments, etc.)

filtered\_tokens = []

for token in tokens:

if not is\_unwanted\_token(token):

filtered\_tokens.append(token)

return filtered\_tokens

function is\_unwanted\_token(token):

// Implement logic to determine if the token is unwanted (e.g., identifier, keyword, comment)

return is\_identifier(token) or is\_keyword(token) or is\_comment(token)

function is\_identifier(token):

// Implement logic to check if the token is an identifier

return match(/[a-zA-Z\_][a-zA-Z0-9\_]\*/, token)

function is\_keyword(token):

// Implement logic to check if the token is a keyword

return token in {'if', 'else', 'while', 'for'}

function is\_comment(token):

// Implement logic to check if the token is a comment

return match(/\/\/.\*|\/\\*[\s\S]\*?\\*\//, token)

**Input:**

#include <stdio.h>

#include <stdbool.h>

#include <ctype.h>

bool is\_whitespace(char c)

{ return c == ' ' || c == '\t' || c == '\n';

}bool is\_operator(char c)

{ return c == '+' || c == '-' || c == '\*' || c == '/' || c == '%' || c == '=' || c == '<' || c == '>';

}bool is\_valid\_identifier\_char(char c)

{ return isalnum(c) || c == '\_';

}void remove\_identifiers(FILE \*input\_file, FILE \*output\_file) {

int c;

bool in\_identifier = false;

while ((c = fgetc(input\_file)) != EOF) {

if (is\_valid\_identifier\_char(c)) {

in\_identifier = true;

} else {

if (in\_identifier) {

fputc(' ', output\_file);

in\_identifier = false; }

fputc(c, output\_file); } }}

int main()

{ FILE \*input\_file, \*output\_file;

char input\_filename[100], output\_filename[100];

printf("Enter the input file name: ");

scanf("%s", input\_filename);

input\_file = fopen(input\_filename, "r");

if (input\_file == NULL)

{ printf("Error in opening input file.\n");

return 1;

} printf("Enter the output file name: ");

scanf("%s", output\_filename);

output\_file = fopen(output\_filename, "w");

if (output\_file == NULL)

{ printf("Error opening output file.\n");

fclose(input\_file);

return 1;

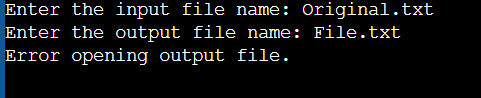
} remove\_identifiers(input\_file, output\_file);

fclose(input\_file);

fclose(output\_file);

printf("Identifiers removed successfully.\n");

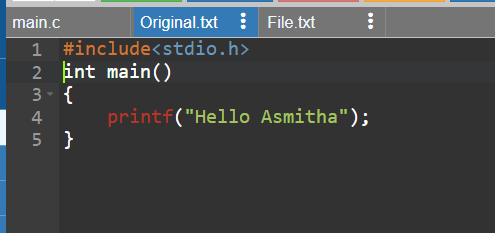
return 0;}

**Output:**

**A screenshot of a computer

Description automatically generated**

**A screen shot of a computer

Description automatically generated**

|  |  |  |  |
| --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment  Department of Computer Science and Engineering  Amity University, Noida (U.P) | | | |
| Programme | B.Tech CSE | Course Name | Complier Construction |
| Course Code | CSE304 | Semester | 6 |
| Student Name |  | Enrollment No. |  |
| Marking Criteria | | | |
| Criteria | Total Marks | Marks Obtained | Comments |
| Concept | 2 |  |  |
| Implementation | 2 |  |  |
| Performance | 2 |  |  |
| Total | 6 |  |  |