UNIVERSITY INSTITUTE OF ENGINEERING AND TECHNOLOGY, PANJAB UNIVERSITY, CHANDIGARH

**COMPILER DESIGN**

****

**Lab File**

Roll No: UE163095

Computer Science and Engineering | 6th SEM

Submitted By: Shivank Bali

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Date** | **Practical** |
| **1** |  | Introduction to Compilers |
| **2** |  | Lexical Analyser |
| **3** |  | Regular Expression to NFA Converter |
| **4** |  |  |
| **5** |  |  |
| **6** |  |  |
| **7** |  |  |
| **8** |  |  |
|  |  |  |

**Practical 1**

Introduction to the compilers

**Compilers**

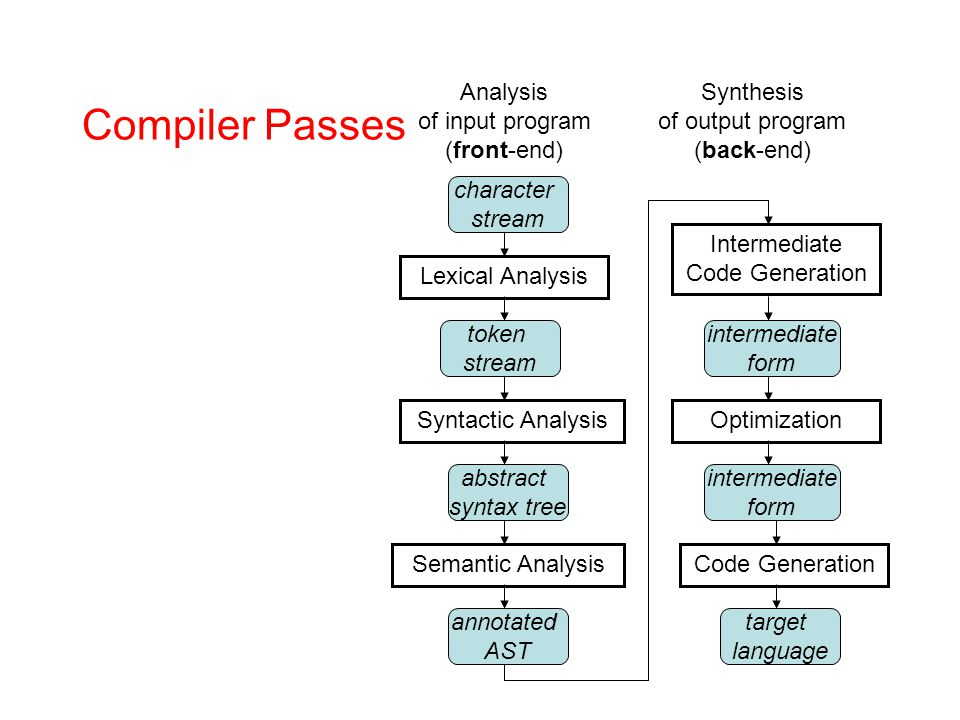
Compiler is a computer programs which transforms the programs/ source codes written in high level languages to the target languages (low level languages). For eg. in case of C language the compiler converts the source code in C language to the Assembly language.

Compilers are a type of translator that support digital devices, primarily computers.

Types Of Compilers:

1. **Native code compiler:** The compiler used to compile a source code for same type of platform only. The output generated by this type of compiler can only be run on the same type of computer system and Os that the compiler itself runs on.
2. **Cross compiler:** The compiler used to compile a source code for different kinds platform. Used in making software’s for embedded systems that can be used on multiple platforms.
3. **Source to source compiler:** the compiler that takes high-level language code as input and outputs source code of another high- level language only.Unlike other compilers which convert high level language into low level machine language, it can take up a code written in Pascal and can transform it into C-conversion of one high level language into another high level language having same type of abstraction . Thus,it is also known as transpiler .
4. **One pass compiler:** It is a type of compiler that compiles the whole process in only one-pass.
5. **Threaded code compiler:** The compiler which simply replace a string by an appropriate binary code.
6. **Incremental compiler:** The compiler which compiles only the changed lines from the source code and update the object code.
7. **Source compiler:** The compiler which converts the source code high level language code in to assembly language only.

Phases of compilers (include description of tokens)



The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



### **Lexical Analysis**

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

### **Syntax Analysis**

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

### **Semantic Analysis**

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

### **Intermediate Code Generation**

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

### **Code Optimization**

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

### **Code Generation**

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

### **Symbol Table**

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

**Compilers vs interpreter**

|  |  |  |
| --- | --- | --- |
| **No** | **Compiler** | **Interpreter** |
| **1** | Compiler Takes **Entire** program as input | Interpreter Takes **Single** instruction as input . |
| **2** | Intermediate Object Code is **Generated** | **No** Intermediate Object Code is **Generated** |
| **3** | Conditional Control Statements are Executes **faster** | Conditional Control Statements are Executes **slower** |
| **4** | **Memory Requirement** : **More** (Since Object Code is Generated) | **Memory Requirement** is **Less** |
| **5** | Program need not be **compiled** every time | Every time higher level program is converted into lower level program |
| **6** | **Errors** are displayed after **entire program** is checked | **Errors** are displayed for **every instruction** interpreted (if any) |
| **7** | **Example** : C Compiler | **Example** : Python |

**Practical 2**

Introduction to Lexical Analyser and its Implementation

**Lexical Analyser**

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

**Lexical Analyser In C++**

#include<iostream>

#include<fstream>

#include<stdlib.h>

#include<string.h>

#include<ctype.h>

using namespace std;

int num\_key = 35;

int isKeyword(char buffer[]){

char keywords[num\_key][10] = { {"auto","break","case","char","const","continue",

"default","do","double","else","enum","extern","float","for",

"goto","if","include","int","long","namespace","register",

"return","short","signed","sizeof","static","struct","switch",

"typedef","union","unsigned","using","void","volatile","while"};

int i, flag = 0;

for(i = 0; i < num\_key; ++i){

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

flag = 1;

break;

}

}

return flag;

}

int main(){

char ch, buffer[15], operators[] = "+-\*/%=";

ifstream fin("code.cpp");

int i,j=0;

if(!fin.is\_open()){

cout<<"error while opening the file\n";

exit(0);

}

while(!fin.eof()){

ch = fin.get();

for(i = 0; i < 6; ++i){

if(ch == operators[i])

cout<<ch<<" is operator\n";

}

if(isalnum(ch)){

buffer[j++] = ch;

}

else if((ch == ' ' || ch == '\n') && (j != 0)){

buffer[j] = '\0';

j = 0;

if(isKeyword(buffer) == 1)

cout<<buffer<<" is keyword\n";

else

cout<<buffer<<" is indentifier\n";

}

}

fin.close();

return 0;

}

**INPUT :**

code.cpp

#include <iostream>

using namespace std;

int main(){

int a , b = 5, c = 4;

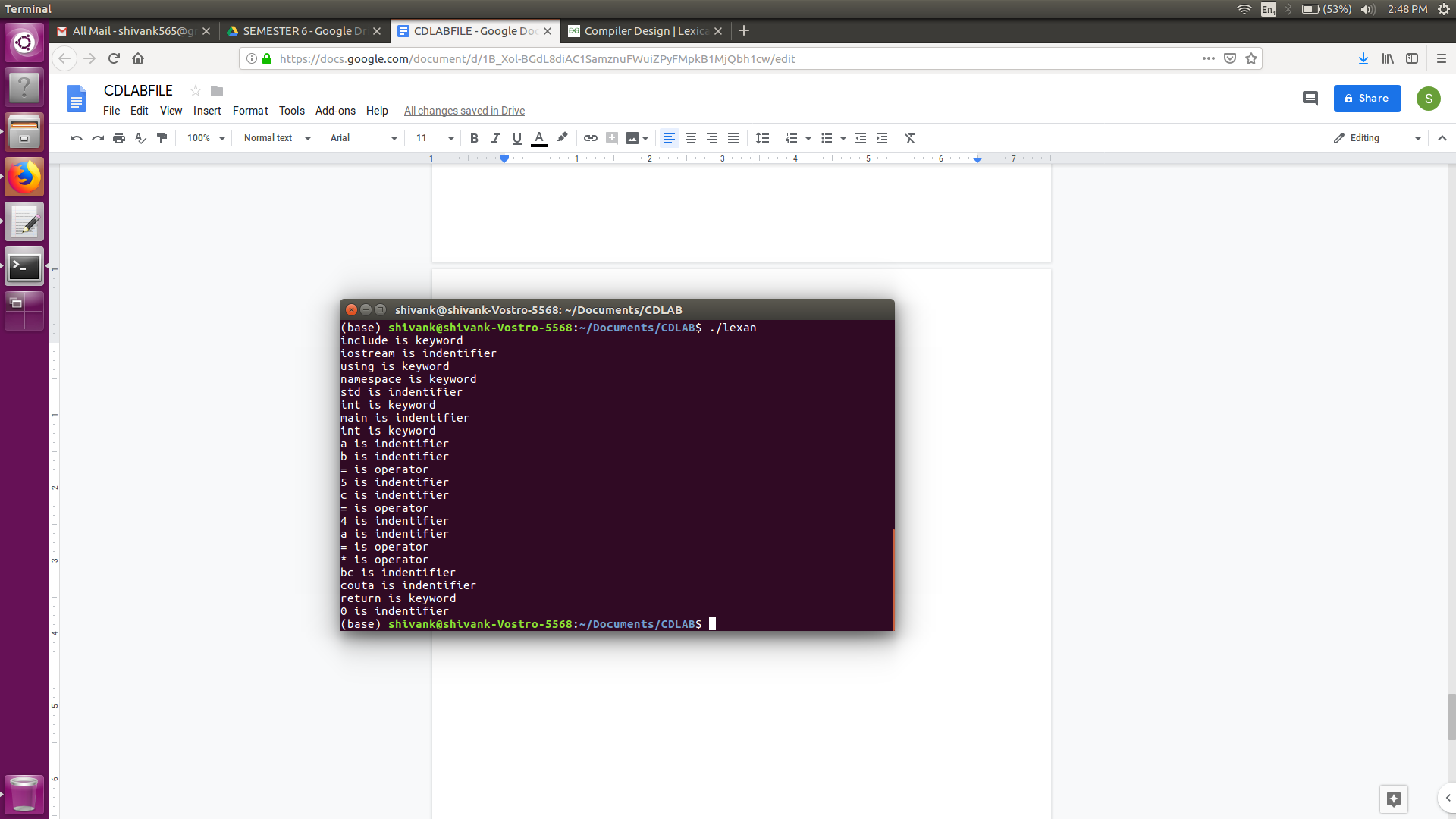
a = b\*c

cout<<a;

return 0;

}

**OUTPUT :**



Practical 3 : Regular Expression to NFA