

**CSL 323 Compiler Construction**

Compiler PROJECT

**GROUP MEMBERS**

Kamal Kumar 02-134201-058

Syed Shaheer Hassan 02-134201-068

Muhammad Hammad Ahmed 02-134201-063

**BS (CS) – 5: SECTION: B**

Department of Computer Science Bahria University, Karachi Campus

**TABLE OF CONTENT**

INTRODUCE

OBJECTIVES

LANGUAGE DESCRIPTION CONTEXT FREE GRAMMAR INPUT

OUPUT

WORK FLOW DAIGRAM INTERMEDIATE CODE REFERENCES

1. **INTRODUCTION**

In this project we have made a compiler similar to C++ using C++ language, all necessary steps are followed. First the input is passed through the lexical analyzer where it is tokenized and then passed on to a syntax analyzer to look for syntax errors and we will ensure whether it follows all rules and is semantically correct

1. **OBJECTIVES**

Following are our objectives.

* + To create a compiler similar to that of C++
  + Because of existing systems, we can easily understand the ideas of the compiler phases.
  + If the input matches the rules the compiler should run it.

1. **LANGUAGE DESCRIPTION:**

* ***Data types***
  + **decimal** (float)
  + **real** (integer)
  + **single** (char)
  + **terms** (string)
  + **dual** (double)
  + **binary(**bool**)**
  + ***Identifier***
  + Starts with **~**
  + Valid identifiers are (**~ \_aa, ~a\_aa,~a\_**)
  + ***Comment***
  + Single Comment (**~~**)
  + Multi line comment (**~/ /~**)
  + ***Loops***
  + Definite loop (same as for loop )

**definite (real ~d = 0; ~d > 10; ~d++){ };**

* + unknown loop (same as while loop )

**unknown ( ~d > 10 ){ };**

* + perform-until loop (same as do-while loop )

**perform {}until{};**

* + ***Conditional Statement***
  + Do-or (same as if else statement)

**do (expression){}or{}**

* + choose (same as switch statement )

**choose (….){option a : …… break; default: break; }**

* + ***Line Terminator***
  + Terminates line with **;**
  + ***Function***

## func a (){ …… }

* + ***Structure***

## Struct name { data types name}

* + ***Array***

## Datatype ~a [2]={“a”,”a”}

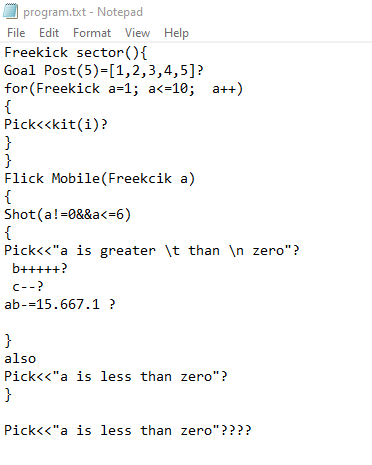
* + ***Input and output statement***
  + Input <- identifier ;
  + Output -> terms\_constant;
  + ***Operators***

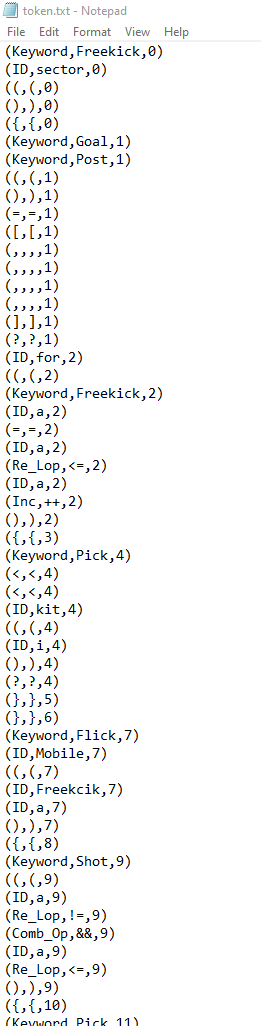
|  |  |
| --- | --- |
| **Arithmetic Operator** | |
| + | addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| % | Modulus |
| **Assignment Operator** | |
| = | Equal to |
| += , =+ | Add + assign or vice versa |
| -=, =- | Subtract + assign or vice versa |
| \*= | Multiply + assign |
| /= | Divide + assign |
| %= | Modulus + assign |
| **Logical Operator** | |
| & | and |
| | | or |
| ! | not |
| **Comparison Operator** | |
| == | Is Equal |
| != | Is not equal |
| < | Is less than |
| > | Is greater than |
| **Input and output operator** | |
| <- | Input |
| -> | output |

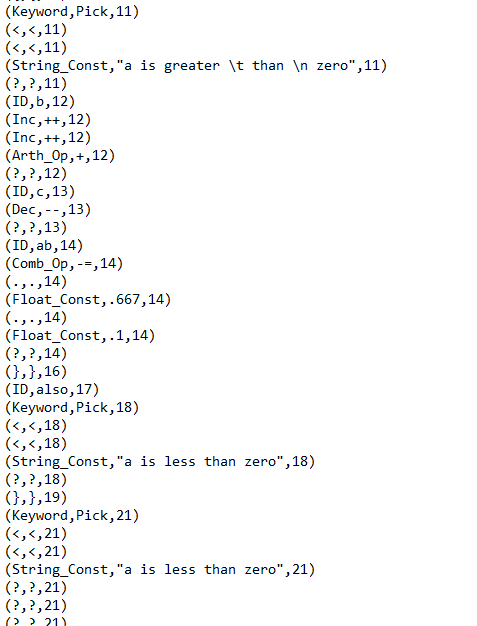
* Keywords

|  |  |  |
| --- | --- | --- |
| **Keyword** | **Keyword** | **Keyword** |
| Real | Choose | Unknown |
| Decimal | Definite | Return |
| Single | Void | Perform |
| Dual | Stop | Until |
| Terms | Main | Do |
| Binary | Default | Or |
| Option | True | Struct |
| Func | Begin | False |

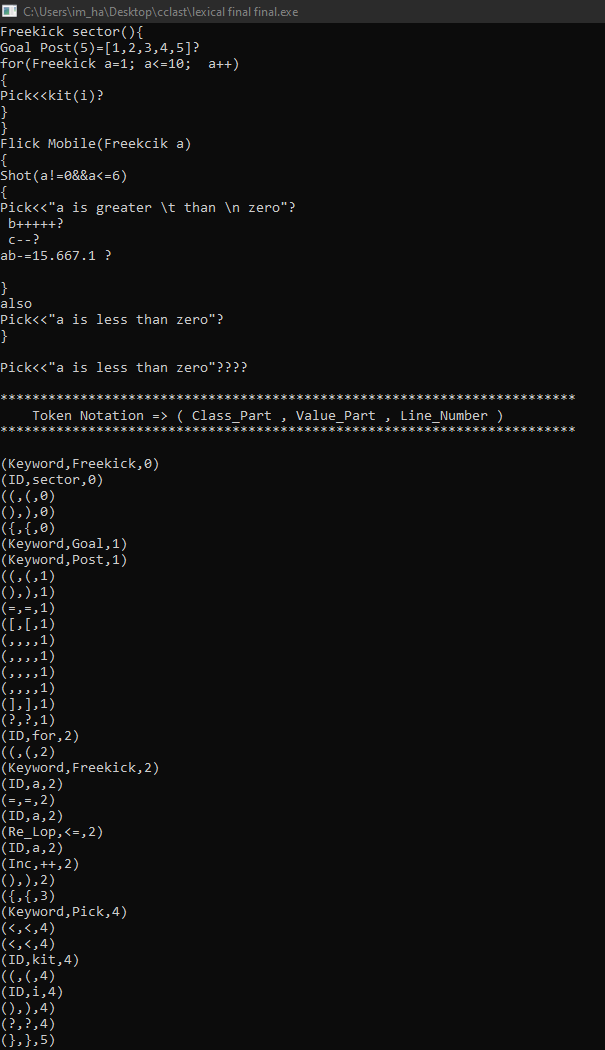
# INPUTS:

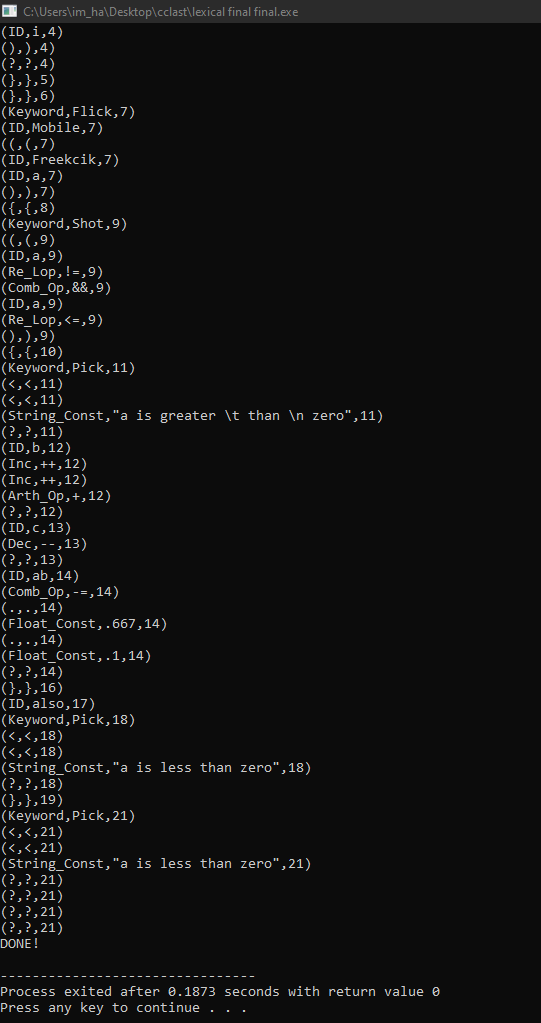
****



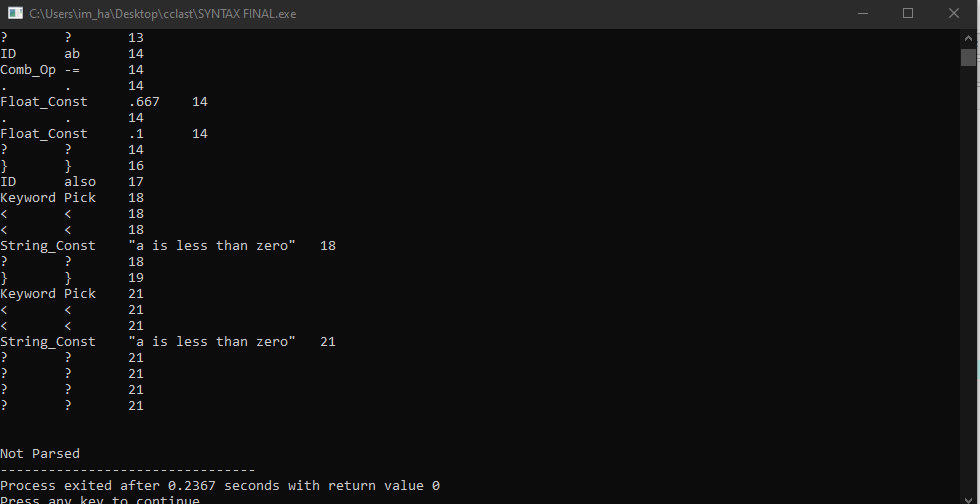


1. **OUTPUTS: Lexical**

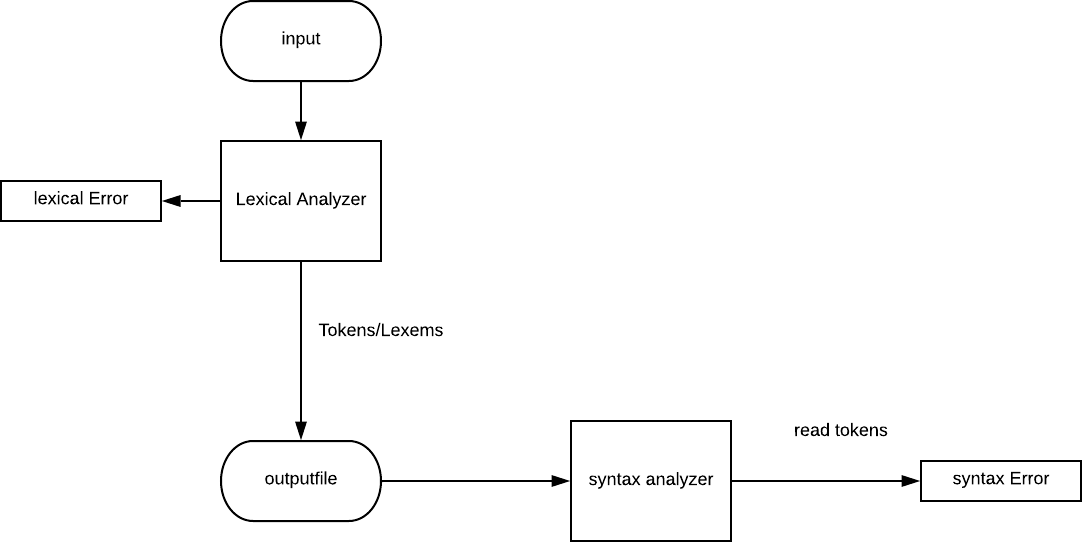




Syntax:



1. **WORK FLOW DIAGRAM**



1. **SEMANTIC ANALYSIS**

The third phase of compiler is sematic analysis, it assure statements as well as declaration in the program are semantically correct. It is an assortment of procedures by which it is used by parser as well as when it is required by the grammar. Syntax tree as well as symbol table are utilized to check the regularity of code. Type checking is a significant piece for semantic analysis where compiler ensures that every administrator has coordinating operands.

Semantic analyzer basically gathers the information and allow it to store in syntax tree else symbol table. This sort data is hence utilized by compiler through intermediate code production.

**Semantic analyzer error**

The **errors** which are recognized by the semantic analyzer are:

1. Mismatch type error
2. No declared variables
3. Misuse of reserved identifier

**Functions**

* **Type checking**

Guarantees that data types are utilized in a manner predictable with their definition.

* **Label checking**

Label references must be contained by the program.

* **Flow of control checking**

It check that control structure is implemented in proper way.

**EXAMPLE**

float a = 40.5; float b = a /4;

With the help the semantic analyzer the integer 4 would be type casted in float 4.0.

There are two type of **semantics Static semantics:**

It is checked at the time of compilation. In the time of execution, the meaning of a program and a static semantics are linked indirectly.

**Dynamic semantics:**

It characterizes the importance of various units of program such as expressions as well as statements. It checks at runtime not at all like static semantics.

1. **INTERMIATE CODE**

Intermediate is used for translating the source code to machine code and intermediate code lies in between the high level language and the low level language

PARSER

Static Checker

Intermediate code

Code generator

Intermediate code generator

o A complete local compiler is necessary for every new machine, if a complier translates source code to the machine code by not having choice for producing intermediate code.

* Synthesis which is the key part of the compiler changes with respect to the target machine.
* With the help of the intermediate code for every machine a new compiler is being eliminated as it keep portion of analysis equal for all compiler.
* By different optimization techniques at intermediate code improves the code performance as it becomes effortless to apply a source code modification.

It is represented in two ways:

**High level IR**

It enhances the performance on the source code as it easy to apply for code modification. It is less preferred when it comes to optimize a target machine.

**Low level IR**

This low level IR is near to the target machine. And the benefit is that it makes it appropriate for register as well as memory allocation etc. Also used in machine dependent optimizations.

# CONTEXT FREE GRAMMER

**<mainfuc>** 🡪 **<beforemain**> main **<type>** start () { **<body>** } **<aftermain>** @

**<type>** 🡪 datatype | void

## <beforemain> 🡪<Struct> <beforemain > | <G\_Decl> <beforemain > |<function>< beforemain >

| **<function\_sign> < beforemain >** | epsilon

<**beforemain>** 🡪 <**function><aftermain>** | epsilon

## <body> 🡪 < multilinestatement >

**< multilinestatement >** 🡪 **< singlelinestatement > < multilinestatement >** |epsilon

**< singlelinestatement >** 🡪 **<Decl>** | **< do\_if\_statement>** | **<unknow>** | **<perform\_until>** |

**<function\_call>** | **<return>** | **<assignment>**

**<return>** 🡪return REAL\_CONST;

**<Struct>** 🡪 struct IDENTIFIER { **<Decl>** } ;

**<Decl>** 🡪 DATATYPE IDENTIFIER **<Decl>’**

## <Decl>’ 🡪 <init><list> | [ <size> ] <A\_init><list>

**<size>** 🡪 **<ID\_int ><size>’** | epsilon

**<size>’** 🡪,**<ID\_int >** | epsilon

**<ID\_int>** 🡪 IDENTIFIER | REAL\_CONST

**<init>** 🡪 ASSIGN\_OP **<init’>** |epsilon

**<init’>** 🡪 IDENTIFIER **<init>** | **<const>** | **<E>**

**<A\_init>** 🡪 ASSIGN\_OP { **<A\_init’ >** | epsilon

## <A\_init’> 🡪 <value> } | <A\_init2> }

**<value>** 🡪 **<E><value1>** |epsilon

**<value1>** 🡪 , **<E><value1>** | epsilon

**<A\_init2>** 🡪 { **<value>** } **<value2>** |epsilon

**<value2>** 🡪 , { **value>** } **<value2>** |epsilon

**<list>** 🡪 ; | , IDENTIFIER **<list’>**

## <list’> 🡪 <init><list> | [ <size> ] <A\_init><list>

**<const>**🡪REAL\_CONST| DECIMAL\_CONST | TERMS\_CONST | SINGLE\_CONST|BINARY\_CONST

**<OE>** 🡪 **<AND>< OE'>**

**<OE'>** 🡪 OR\_OP **< AND >< OE'>** | epsilon

## <AND> 🡪 <RE>< AND'>

**<AND'>** 🡪 AND\_OP **<RE>< AND'>** | epsilon

## <RE> 🡪 <E><RE'>

**<RE'>** 🡪 RELAT\_OP **<E><RE'>** | epsilon

## <E> 🡪 <T><E'>

**<E'>** 🡪 Add\_Sub\_OP **<T><E'>** | epsilon

## <T> 🡪 <F><T'>

**<T'>** 🡪 **<MDM> <F><T'>** | epsilon

**<F>** 🡪 IDENTIFIER **<F>’** | **<const>** | ( **<OE>** ) | Inc\_Dec\_OP ID | NOT\_OP **<F>**

**<F>’** 🡪 ( **<prmtrs2>** ) | Inc\_Dec\_OP |epsilon

**<MDM>** 🡪 Mul\_Div\_OP | MOD\_OP

## <unknown> 🡪 ( <OE><AI> ) <body2>

**<AI>** 🡪 ; **<inc>** | ; **<assign>** |epsilon

**<assign>** 🡪 IDENTIFIER Inc\_Dec\_ASSIGN\_OP **<ID\_int>**

**<inc>** 🡪 IDENTIFIER Inc\_Dec\_OP | Inc\_Dec\_OP IDENTIFIER

**<body2>** 🡪 ; | **<sst>** | { **<mst>** }

## <do\_if\_statement> 🡪 do( <OE> ) <body2><or\_else>

**<or\_else>** 🡪 or **<body2>** | epsilon

**<function>** 🡪 func **<type>** IDENTIFIER ( **<prmtrs >** ) { **<body>** }

**<type>** 🡪 void | DATATYPES

**< prmtrs>** 🡪 DATATYPE IDENTIFIER **<m\_prmtrs>** | epsilon

## <m\_prmtrs> 🡪 , DATATYPE IDENTIFIER <m\_prmtrs> | epsilon

**<func\_sign>** 🡪 **<type>** IDENTIFIER ( **<prmtrs >** );

**<func\_call>** 🡪IDENTIFIER ( **<prmtrs2>** );

**<prmtrs2>** 🡪 **<E><m\_prmtrs2>** | epsilon

**<m\_prmtrs2>** 🡪 , **<E> <m\_prmtrs2>** | epsilon

## <G\_delc> 🡪 <Decl>

**<perform\_until>** 🡪 perform { **<body>** } until( **<OE>** );

**<assignment>** 🡪 IDENTIFIER ASSIGN\_OP **<E>** ;

**<choosestatemnt>**🡪choose(IDENTIFIER):{<**option**>}

**<option>**🡪option : IDENTIFIER **<option> |**

# REFERENCES

1. <https://github.com/topics/compiler-design>
2. <https://github.com/topics/compiler-construction>
3. <https://www.thecrazyprogrammer.com/2017/02/lexical-analyzer-in-c.html>
4. [https://www.codepoc.io/blog/cpp/3028/program-to-implement-lexical- analyzer?utm\_source=17](https://www.codepoc.io/blog/cpp/3028/program-to-implement-lexical-analyzer?utm_source=17)