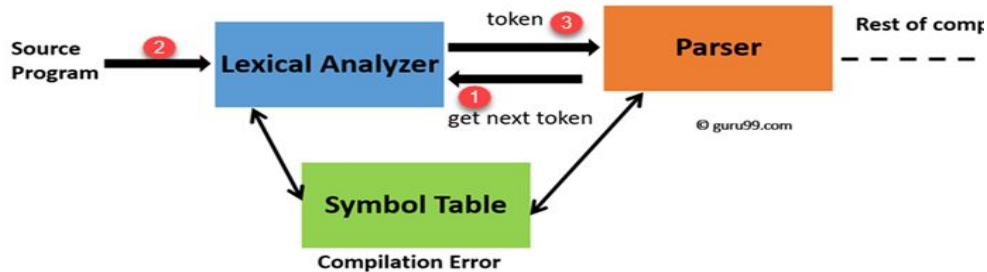


# Tureasy - Compiler Design Overview

Manoj Gayala	CS19BTECH11011
Sravanthi M	ES19BTECH11007
Nisha M	CS19BTECH11012
Sharanya	CS19BTECH11020
Gupta	CS19BTECH11042
Suraj Gubbala	CS19BTECH11057
Vedika Verma	CS19BTECH11036
Jatin Kumar	

# The Role of lexical Analyzer

- It is the first Phase of Compiler Design.
- Basically it reads user input and produces a sequence tokens that is used in Syntax analysis by parser.



# Ocaml for lexical analysis



- We used Ocaml language to write our lexer. The main advantage of using it is, we would be able to parse our complex data structure formats and hence being efficient in providing useful error messages which becomes a complex task otherwise.
- We have written the code for lexer in Ocaml in the file named `lexer.mll` . We execute the file with the command **`ocamllex lexer.mll`** that produces a lexical analyzer from a set of regular expressions with attached semantic actions, in the style of `lex` in the file **`lexer.ml`**.
- This file defines one lexing function per entry point in the lexer definition. These functions have the same names as the entry points. Lexing functions take as argument a lexer buffer, and return the semantic attribute of the corresponding entry point.

# Overview of lexer.mll file



- This file contains the code written in Ocaml to parse the tokens (syntax , keywords, datatypes etc..)
- To give a brief idea of the non primitive datatypes that we added specific to our language include
  - | "matrix" { MATRIX }
  - | "graph" { GRAPH }
  - | "numset" { NUMSET }
  - | "strset" { STRSET }
- The tags implementation in our language as mentioned in the whitepaper requires additional tokens as below:
  - | "#" (letter(letter|digit)\* as tag) { TAG\_BEGIN(tag) }
  - | "#!" (letter(letter|digit)\* as tag) { TAG\_END(tag) }
- This was the tokenisation part of our lexical analyser where the output briefly corresponds to set of strings(Identifiers, keywords, whitespace,Integer).

# Working of lexer

## Correct Syntax



A screenshot of a text editor window titled "testing.tz" with the path "~/compilers-2-project-team-8-aug21/de...". The editor contains the following C code:

```
1 struct abc {  
2   matrix M;  
3   $ graph c;  
4 };  
5  
6 int a = 0, b = 1+4;  
7 $*  
8 string message = "hello!";  
9 *$
```

The status bar at the bottom indicates "Plain Text", "Tab Width: 8", "Ln 9, Col 3", and "INS".

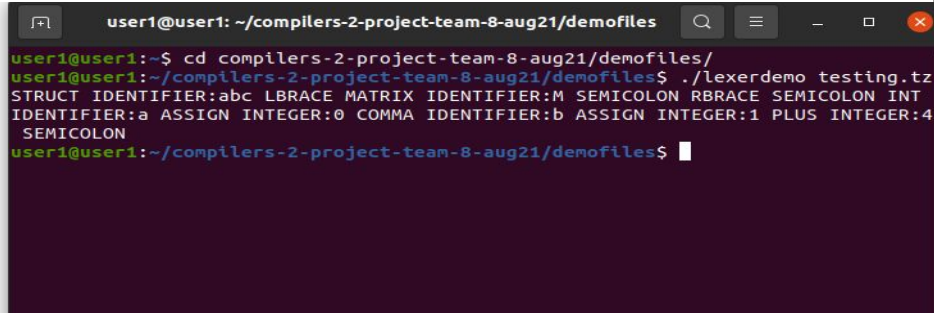
## Incorrect Syntax



A screenshot of a text editor window titled "testing.tz" with the path "~/compilers-2-project-team-8-aug21/de...". The editor contains the following C code:

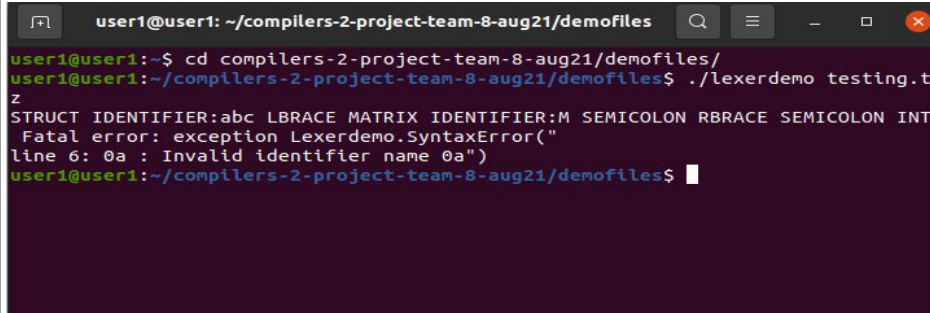
```
1 struct abc {  
2   matrix M;  
3   $ graph c;  
4 };  
5  
6 int 0a = 0, b = 1+4;  
7 $*  
8 string message = "hello!";  
9 *$
```

The status bar at the bottom indicates "Plain Text", "Tab Width: 8", "Ln 6, Col 6", and "INS".



A screenshot of a terminal window with the prompt "user1@user1: ~/compilers-2-project-team-8-aug21/demofiles". The user has executed the command `./lexerdemo testing.tz`. The output shows the tokens for each line of the correct code:

```
user1@user1:~/compilers-2-project-team-8-aug21/demofiles/$ cd compilers-2-project-team-8-aug21/demofiles/  
user1@user1:~/compilers-2-project-team-8-aug21/demofiles$ ./lexerdemo testing.tz  
STRUCT IDENTIFIER:abc LBRACE MATRIX IDENTIFIER:M SEMICOLON RBRACE SEMICOLON INT  
IDENTIFIER:a ASSIGN INTEGER:0 COMMA IDENTIFIER:b ASSIGN INTEGER:1 PLUS INTEGER:4  
SEMICOLON  
user1@user1:~/compilers-2-project-team-8-aug21/demofiles$
```



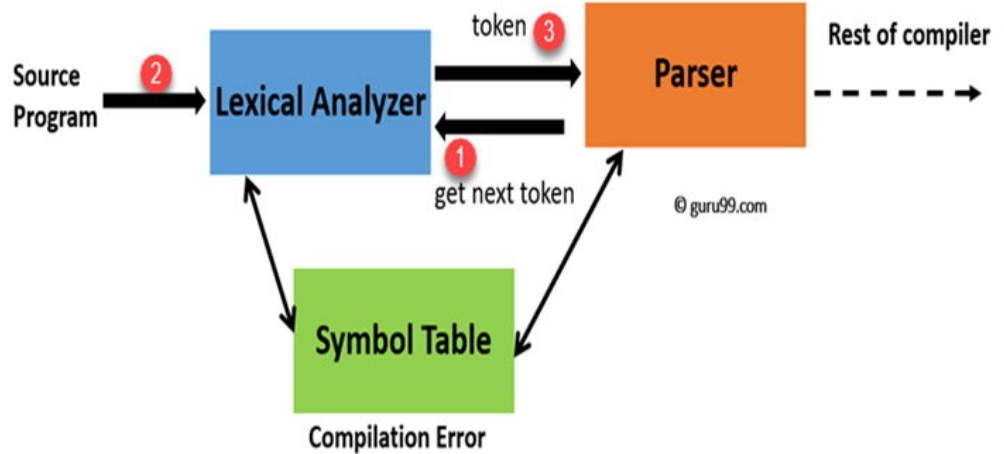
A screenshot of a terminal window with the prompt "user1@user1: ~/compilers-2-project-team-8-aug21/demofiles". The user has executed the command `./lexerdemo testing.tz`. The output shows the tokens for the first five lines, followed by a fatal error on line 6:

```
user1@user1:~/compilers-2-project-team-8-aug21/demofiles/$ cd compilers-2-project-team-8-aug21/demofiles/  
user1@user1:~/compilers-2-project-team-8-aug21/demofiles$ ./lexerdemo testing.tz  
STRUCT IDENTIFIER:abc LBRACE MATRIX IDENTIFIER:M SEMICOLON RBRACE SEMICOLON INT  
z  
Fatal error: exception Lexerdemo.SyntaxError("line 6: 0a : Invalid identifier name 0a")  
user1@user1:~/compilers-2-project-team-8-aug21/demofiles$
```



# The Role of Parser

- The parser obtains a string of tokens from the lexical analyser, and verifies that the string of token names can be generated by the **grammar** for the source language. Report any syntax errors and also **recover** from common errors.
- The **parser constructs a parse tree** and passes it to the rest of the compiler for further processing.



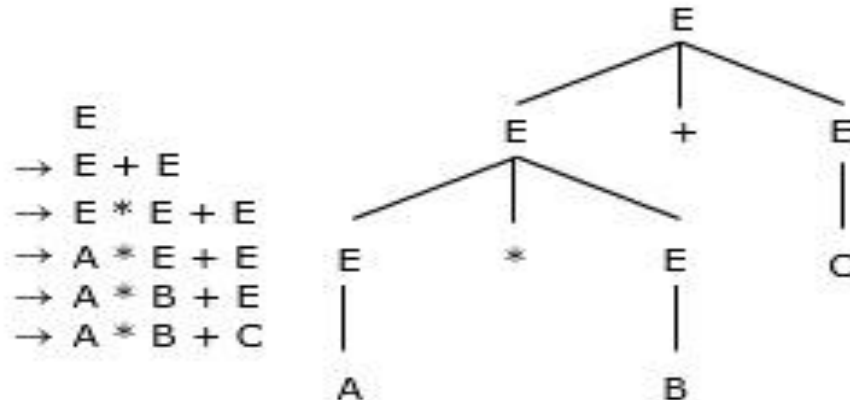
## What is parsing

1. Parsing is used to derive using the production rules of grammar.
2. It is used to check the acceptability of given string.
3. A parse tree is a **graphical representation** of a derivation

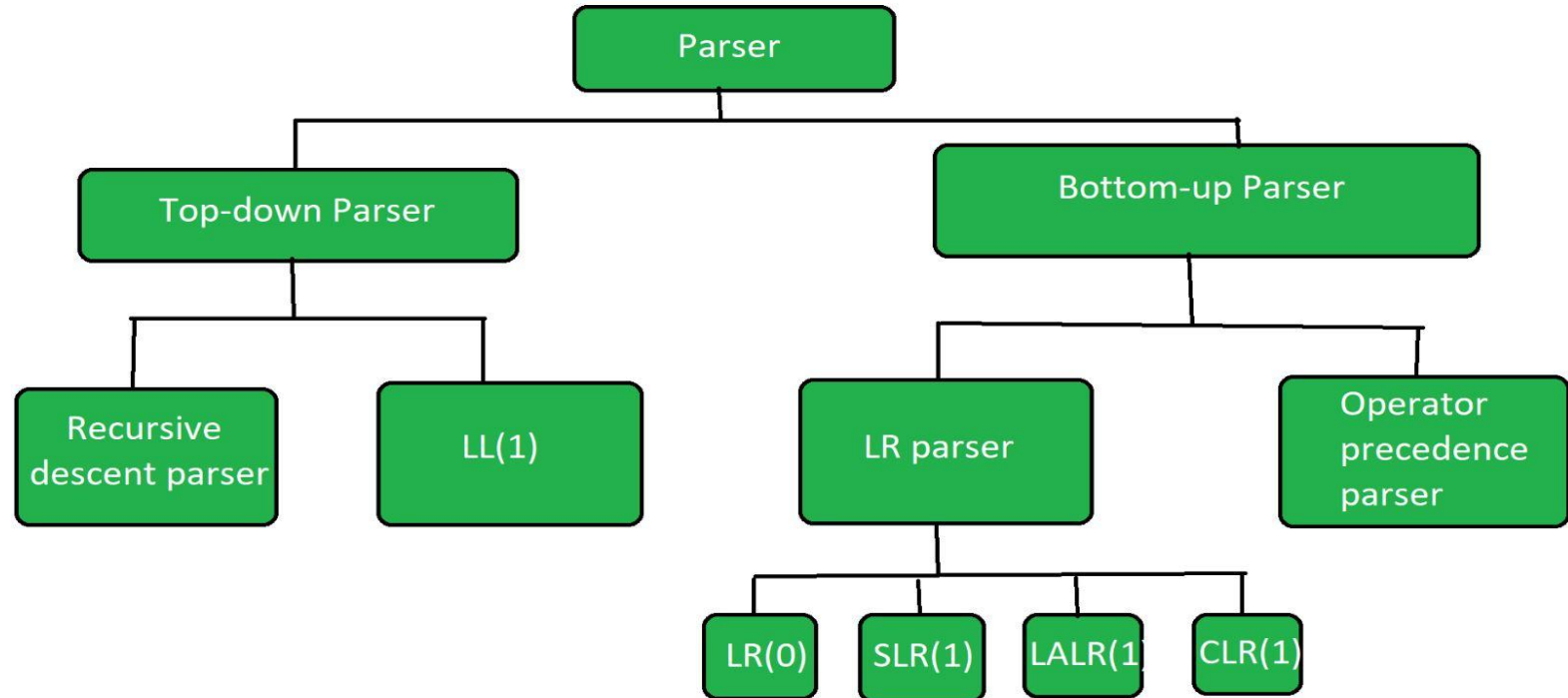
$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

Example-

Here is an example of parse tree.



# Types of Parser-





## Top-Down Parsing -

It is a type of parsing in which parser starts constructing tree from start symbol and then tries to change start symbol to the input.

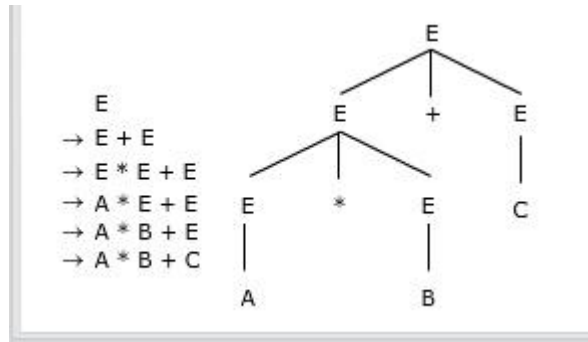
### Recursive Descent parsing-

It uses recursive methods to process the input.

### Backtracking-

This method may take input string more than once to give correct output.

### Examples-

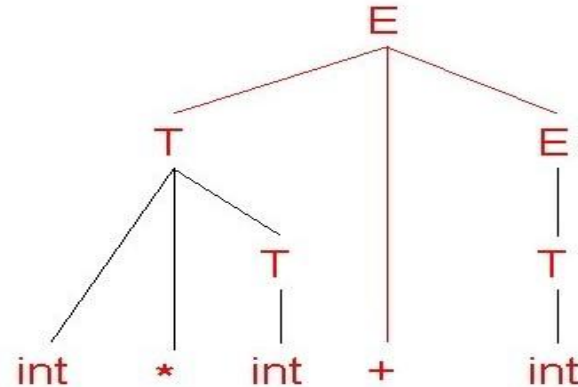


## Bottom-Up parser-

- It is a parser in which Parsing starts from the bottom with string input and comes to the start symbol using parse tree.
- It uses stack to store both sequential and state forms.
- It reduces a sequences of tokens to the start symbol.

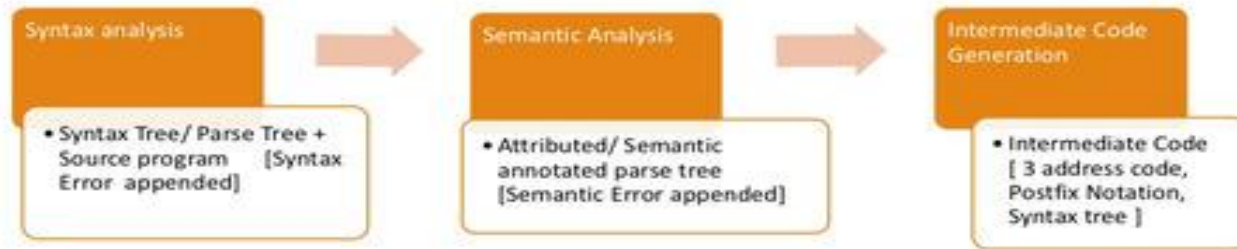
### Example-

int \* int + int  
int \* T + int  
T + int  
T + T  
T + E  
E

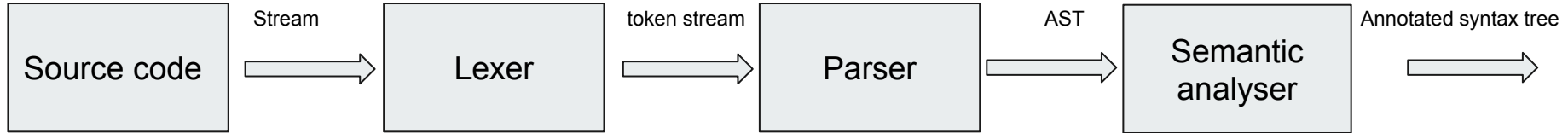


# Semantic Analysis


In this phase we ensure that the declarations and statements of a program are semantically correct. That is that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.



# Pipeline:



# What does a semantic analyser do?

-  Type checking: To make it sure that the operator is applied to compatible operands
- Label Checking: Labels references in a program must exist.
- Flow control checks: Control structures must be used in their proper fashion.
- Uniqueness Checking: Make it sure unique declaration of variable in a scope.
- Scope resolution

# Semantic errors:



- Type mismatch
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Function overloading
- Actual and formal parameter mismatch.
- Unknown function calls
- Undeclared identifiers
- Errors in non primitive data types:
  - Matrix: different row sizes, wrong index type
  - Struct: empty declaration, illegal member access

## Example 1: Empty struct declaration

```
int a;  
  
struct x {  
}  
  
def void main ()  
{  
    int b;  
    return;  
}
```

```
(base) sharanya@sharanya-Swift-SF314-55G:~/Documents/compilers-2-project-team-8-aug21-main(1)/compilers-2-project-team-8-aug21-main$ ./a.out testcases/fail_emptystruct.tz  
Fatal error: exception Failure("Found struct without fields 'x'")
```

## Example 2: Attempting to declare built-in function

```
def void print (int x)  
{  
    int y;  
}
```

```
gayalamanoj@manojgayala:~/Desktop/SEMESTER 5/COMPILERS/compilers-2-project-team-8-aug21-main$ ./xyz test.tz  
Fatal error: exception Failure("Function print cannot be defined, it is built-in")
```

### Example 3: No main

```
def int incr(int a)
{
    return a+1;
}
```

```
(base) sharanya@sharanya-Swift-SF314-55G:~/Documents/compilers-2-project-team-8-aug21-main(1)/compilers-2-project-team-8-aug21-main$ ./a.out testcases/func4.tz
Fatal error: exception Failure("Unknown function main call")
```

### Example 4: Duplicate detection

```
int x;
int x;
```

```
gayalamanoj@manojgayala:~/Desktop/SEMESTER 5/COMPILERS/compilers-2-project-team-8-aug21-main$ ./xyz test.tz
Fatal error: exception Failure("Found duplicate global 'x'")
```



# Code generation phase



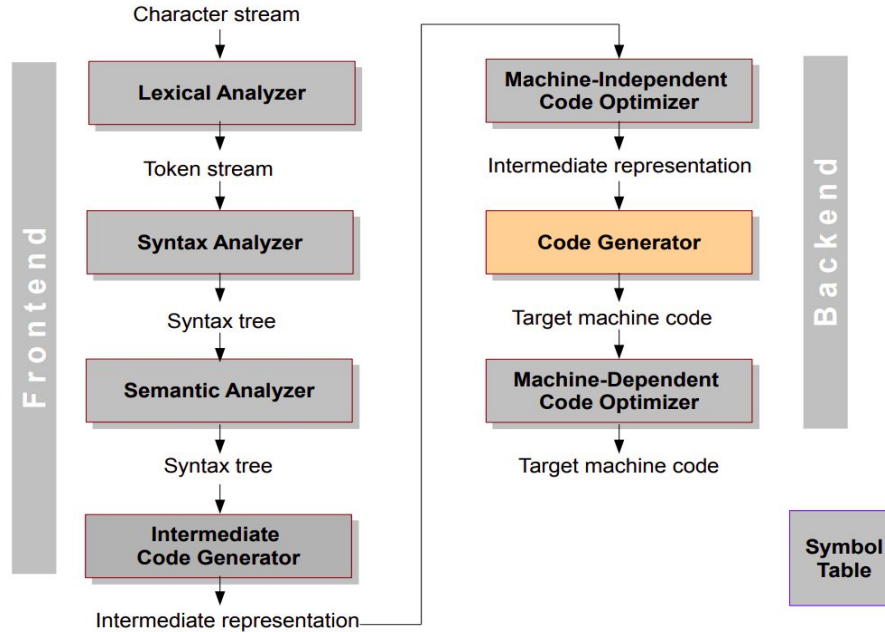
- It's the last stage of compiler development.
- This phase generates assembly code, which is our target code.
- This lower level object code generated has the following properties :
  - It carries the exact meaning of the source code.
  - It would be efficient in terms of CPU usage and memory management.

## **Includes -**

- Allocation of memory to each variable.
- The instructions are broken down into a series of assembly instructions.
- Memory registers are used to store variables and intermediated results.

The code generator itself should run efficiently while performing the above actions.

# Front end and back end



Involvement of symbol table - Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.



- In reality , the problem of generating the target code is undecidable . The problems like register allocation etc remain NP hard.
- Elaboration of the three basic steps in code generation
  - Instruction Selection - The complexity of this depends upon the level of IR and the desired quality of the generated code
  - Register allocation -
    - Allocation - which variables should be put into registers .
    - Assignment - which register should be used for a variable .
    - Finding an optimal assignment of registers to variables is NP complete.
  - Instruction ordering - Picking the best order is NP complete . The code generator has to look at multiple instructions at a time.