Theory of Computation (CT-502)

Course Instructor
ANUJ GHIMIRE

DISCLAIMER

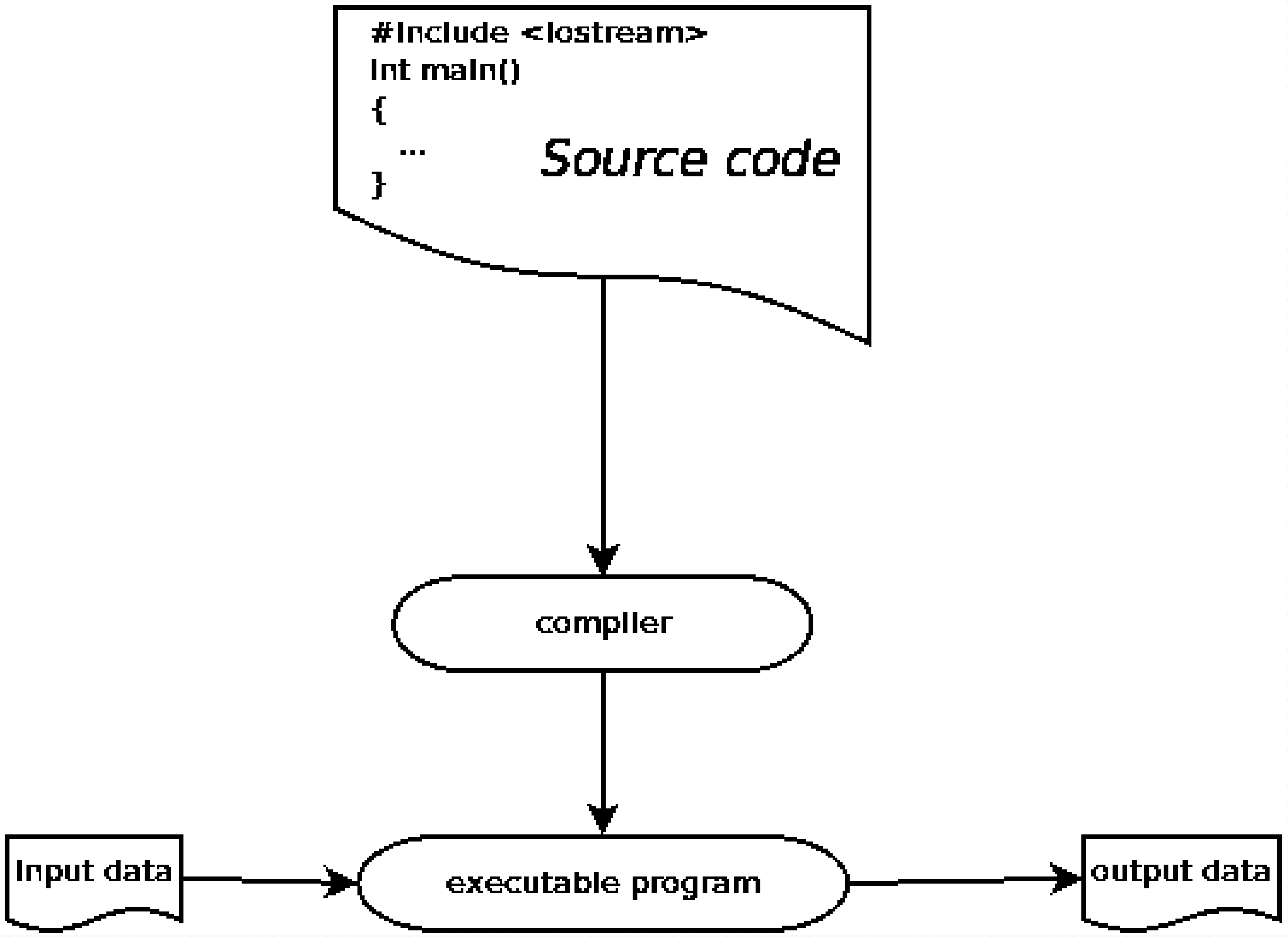
Chapter 6

Automata Theory and Compiler Design

Compiler

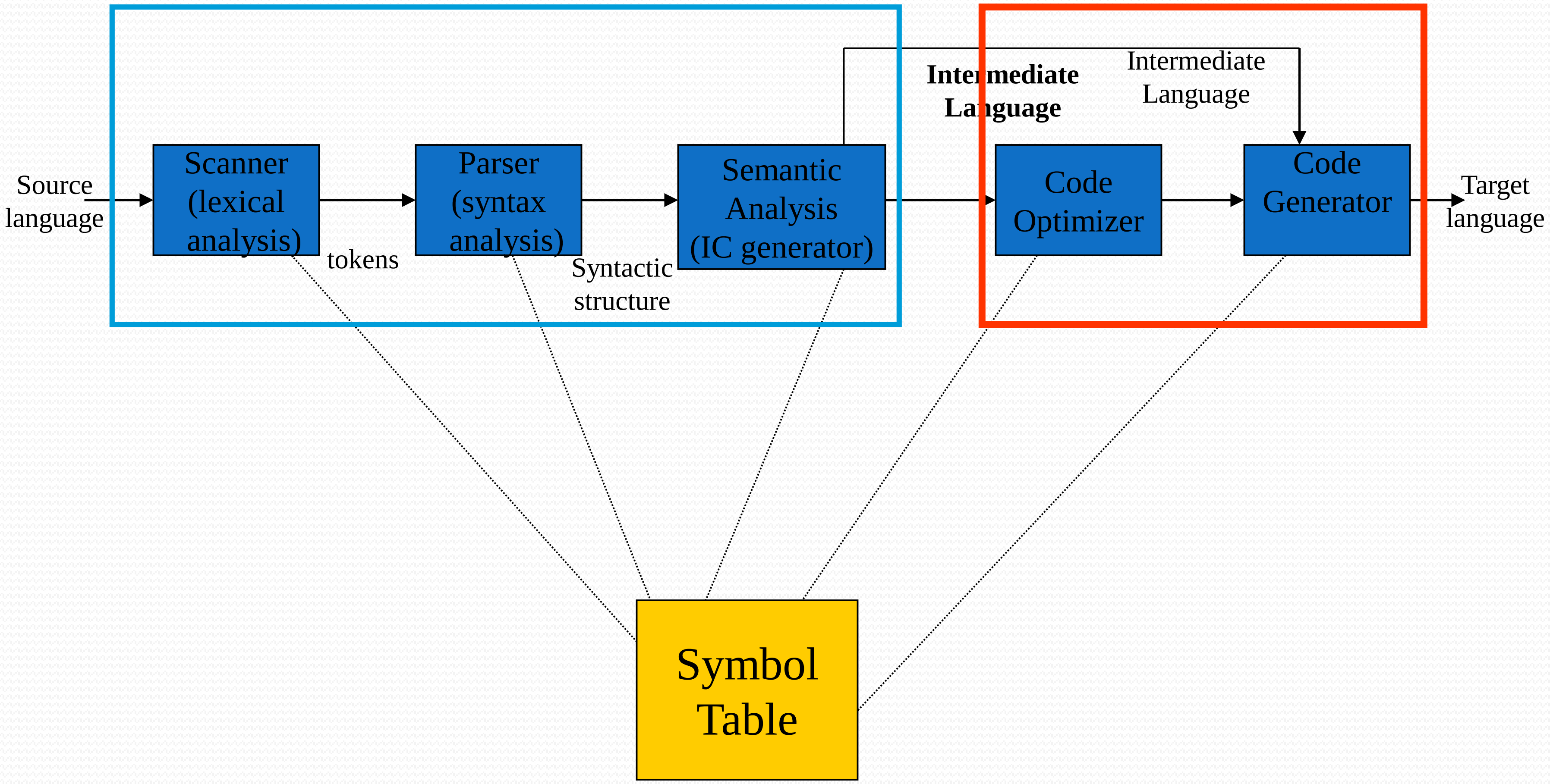
- Compiler is a translator program that translates a program written in (HLL) the ***source program*** and translates it into an equivalent program in (MLL) the ***target program.***
- As an important part of a compiler is ***error showing to the programmer.***
- Executing a program written in HLL programming language is basically of two parts
  - Source program must first be compiled translated into a object program
  - Then the results object program is loaded into a memory executed.

Compiler

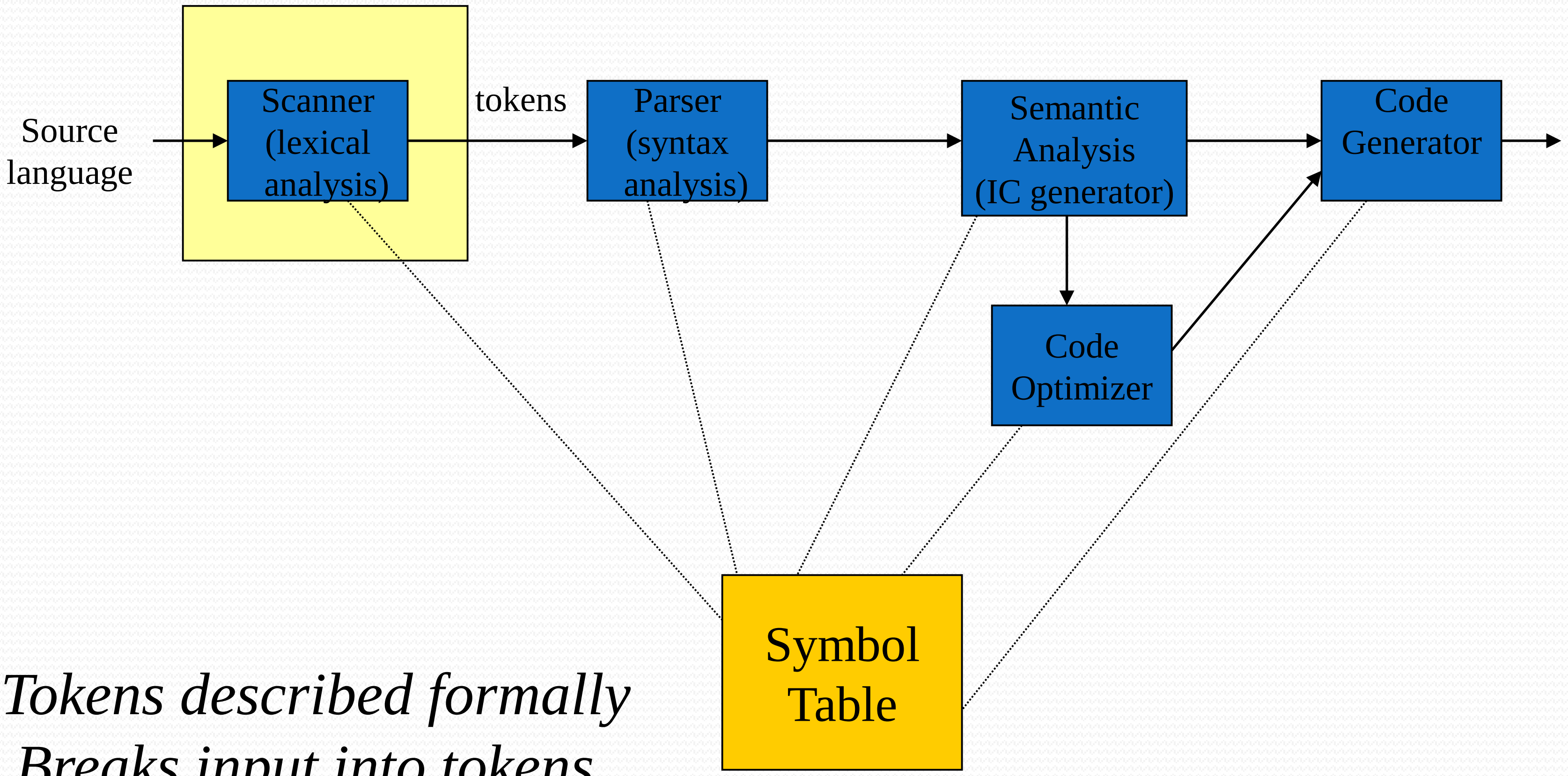

#include <iostream>
int main()
{
...
}
Source code
compiler
input data
executable program
output data

Phases of Compiler

- A compiler operates in phases.
- A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation.
- There are two phases of compilation.
  - Analysis (Machine Independent/Language Dependent)
  - Synthesis (Machine Dependent/Language independent)
- Compilation process is partitioned into no-of-sub processes called ‘***phases***’


Source language
Scanner (lexical analysis)
tokens
Parser (syntax analysis)
Syntactic structure
Semantic Analysis (IC generator)
Intermediate Language
Code Optimizer
Intermediate Language
Code Generator
Target language
Symbol Table

Lexical Analysis



- *Tokens described formally*
- *Breaks input into tokens*
- *Remove white space*

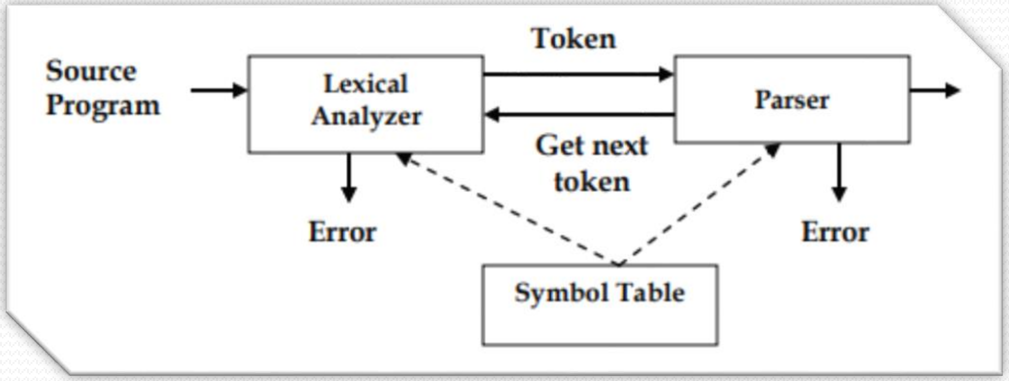
Lexical Analysis

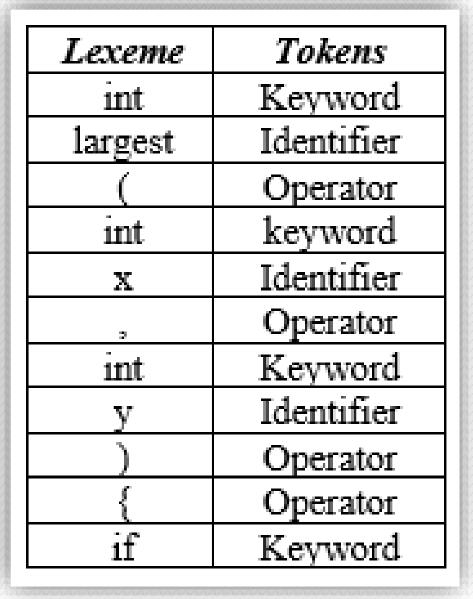
- The lexical analyzer reads the input characters of the source program, groups them into lexemes, and produces a sequence of tokens for each lexeme.
- The tokens are then sent to the parser for syntax analysis.
- Normally a lexical analyzer doesn't return a list of tokens; it returns a token only when the parser asks a token from it.
- Lexical analyzer may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc.

Lexical Analysis

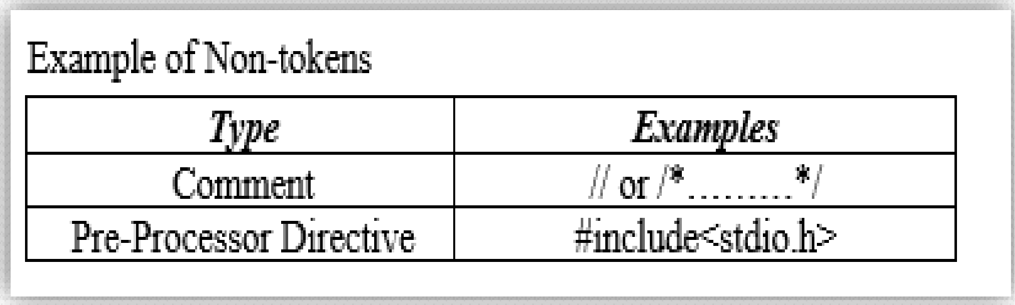
Consider the following code that is fed to Lexical Analyzer

```
#include <stdio.h>
int largest(int x, int y)
{
   if (x > y)
   return x;
   else
   return y;
}
```





Lexeme	*Tokens*
int	Keyword
largest	Identifier
(	Operator
int	keyword
x	Identifier
,	Operator
int	Keyword
y	Identifier
)	Operator
{	Operator
if	Keyword



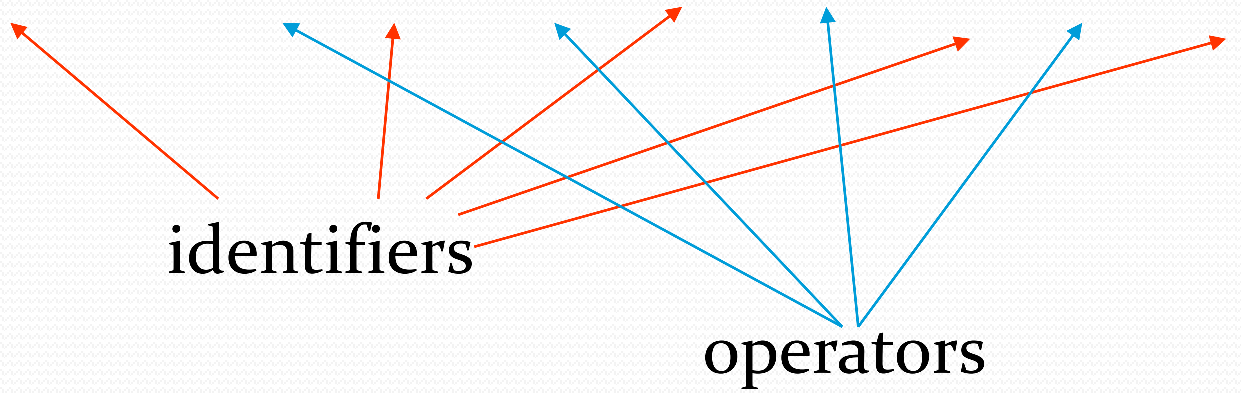
Example of Non-tokens

Type	*Examples*
Comment	// or /*.........*/
Pre-Processor Directive	#include<stdio.h>

Lexical Analysis

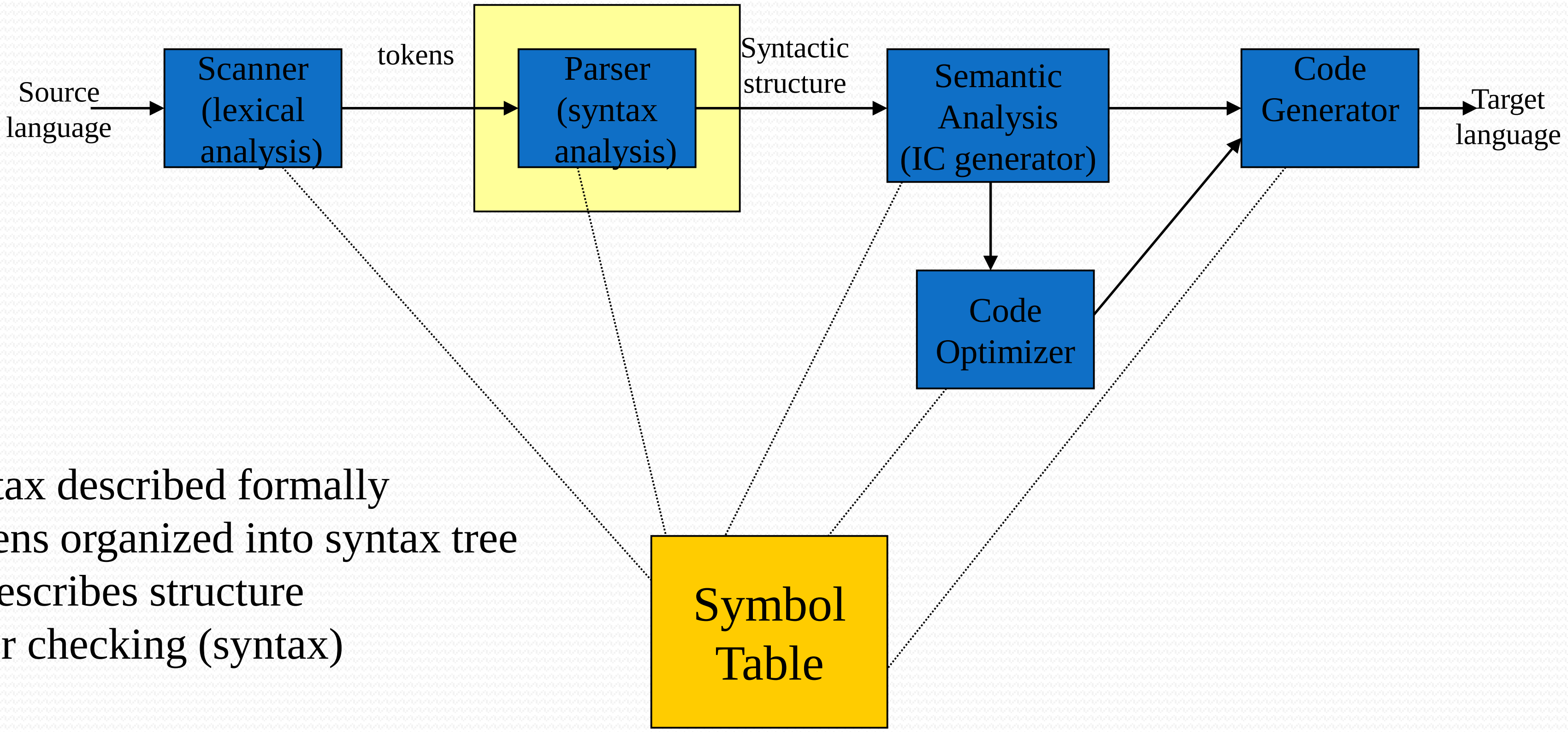
Tokens:

‘result’, ‘=‘, ‘a’, ‘+’, ‘b’, ‘*’, ‘c’, ‘/’, ‘d’



Input: result = a + b * c / d

Syntax Analysis



- Syntax described formally
- Tokens organized into syntax tree that describes structure
- Error checking (syntax)

Syntax Analysis

- Once lexical analysis is completed the generation of lexemes and mapped to the token, then parser takes over to check whether the sequence of tokens is grammatically correct or not, according to the rules that define the syntax of the source language.
- The main purposes of Syntax analyzer are:
  - *Syntax analyzer is capable analyzes the tokenized code for structure.*
  - *This is able to tags groups with type information.*
- A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given source program. Syntax analyzer is also called the parser.

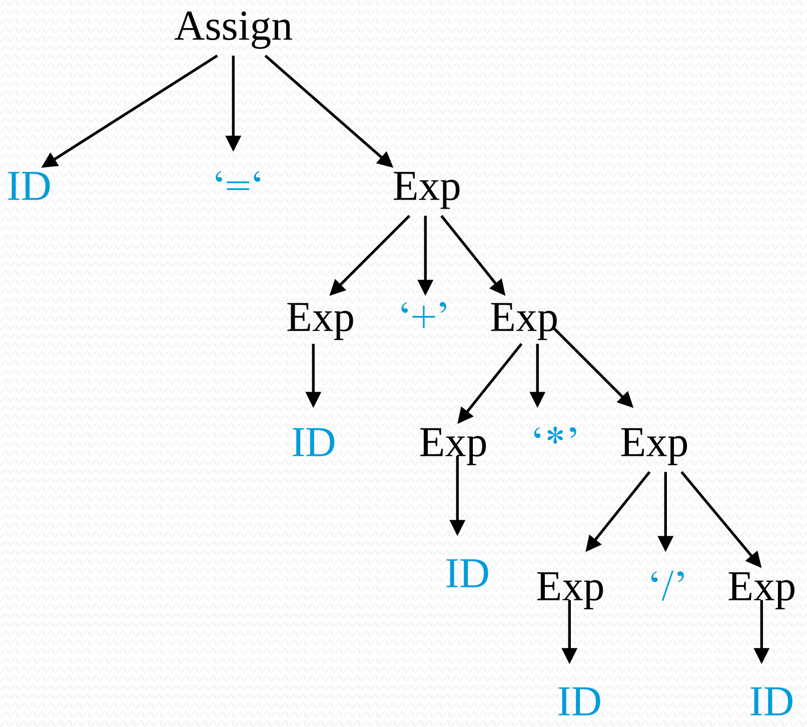
Syntax Analysis

- Its job is to analyze the source program based on the definition of its syntax.
- It is responsible for creating a parse-tree of the source code.

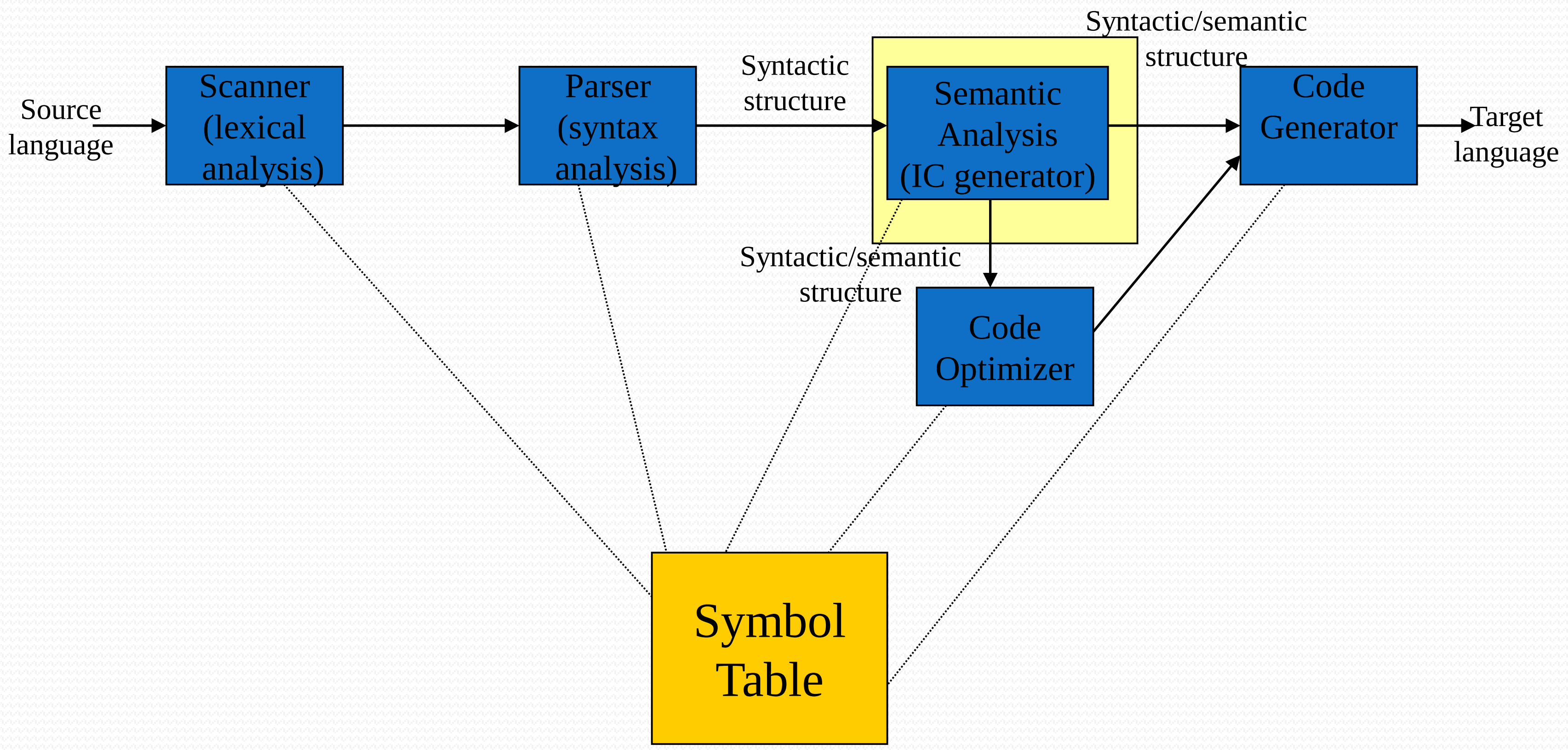
Syntax Analysis

Input: result = a + b * c / d

Exp	::=	Exp '+' Exp
	|	Exp '-' Exp
	|	Exp '*' Exp
	|	Exp '/' Exp
	|	ID
Assign	::=	ID '=' Exp



Semantic Analysis


Source language
Scanner (lexical analysis)
Parser (syntax analysis)
Syntactic structure
Semantic Analysis (IC generator)
Syntactic/semantic structure
Code Generator
Target language
Syntactic/semantic structure
Code Optimizer
Symbol Table

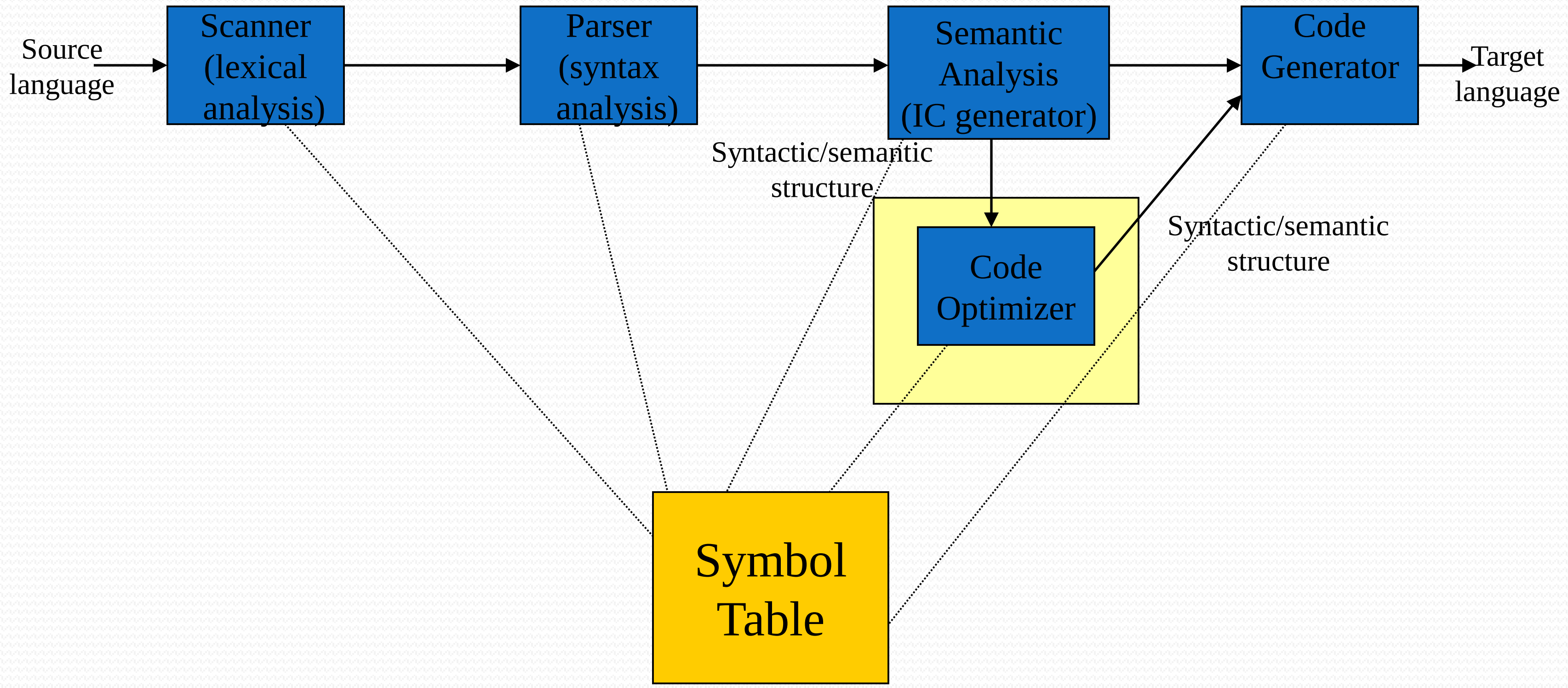
Semantic Analysis

- The next phase of the semantic analyzer is the semantic analyzer and it performs a very important role to check the semantics rules of the expression according to the source language.
- The parsing phase only verifies that the program consists of tokens arranged in a syntactically valid combination.
- Now semantic analyzer checks whether they form a sensible set of instructions in the programming language or not.

Semantic Analysis

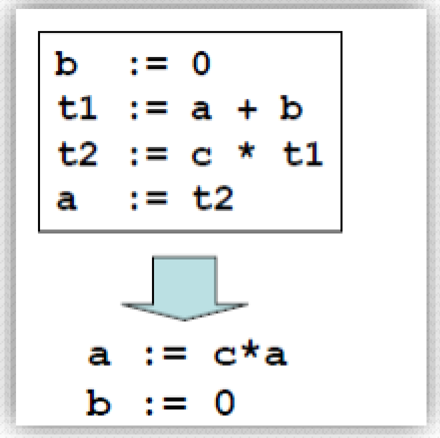
- Some examples of the things checked in this phase are listed below:
  - *The type of the right side expression of an assignment statement should match the type of the left side ie. in the expression* ***newval = oldval + 12****, the type of the expression* ***(oldval+12)*** *must match with type of the variable* ***newval.***
  - *The parameter of a function should match the arguments of a function call in both number and type.* *
  - *The variable name used in the program must be unique etc.*

Code Optimization


Source language
Scanner (lexical analysis)
Parser (syntax analysis)
Semantic Analysis (IC generator)
Code Generator
Target language
Syntactic/semantic structure
Code Optimizer
Syntactic/semantic structure
Symbol Table

Code Optimization

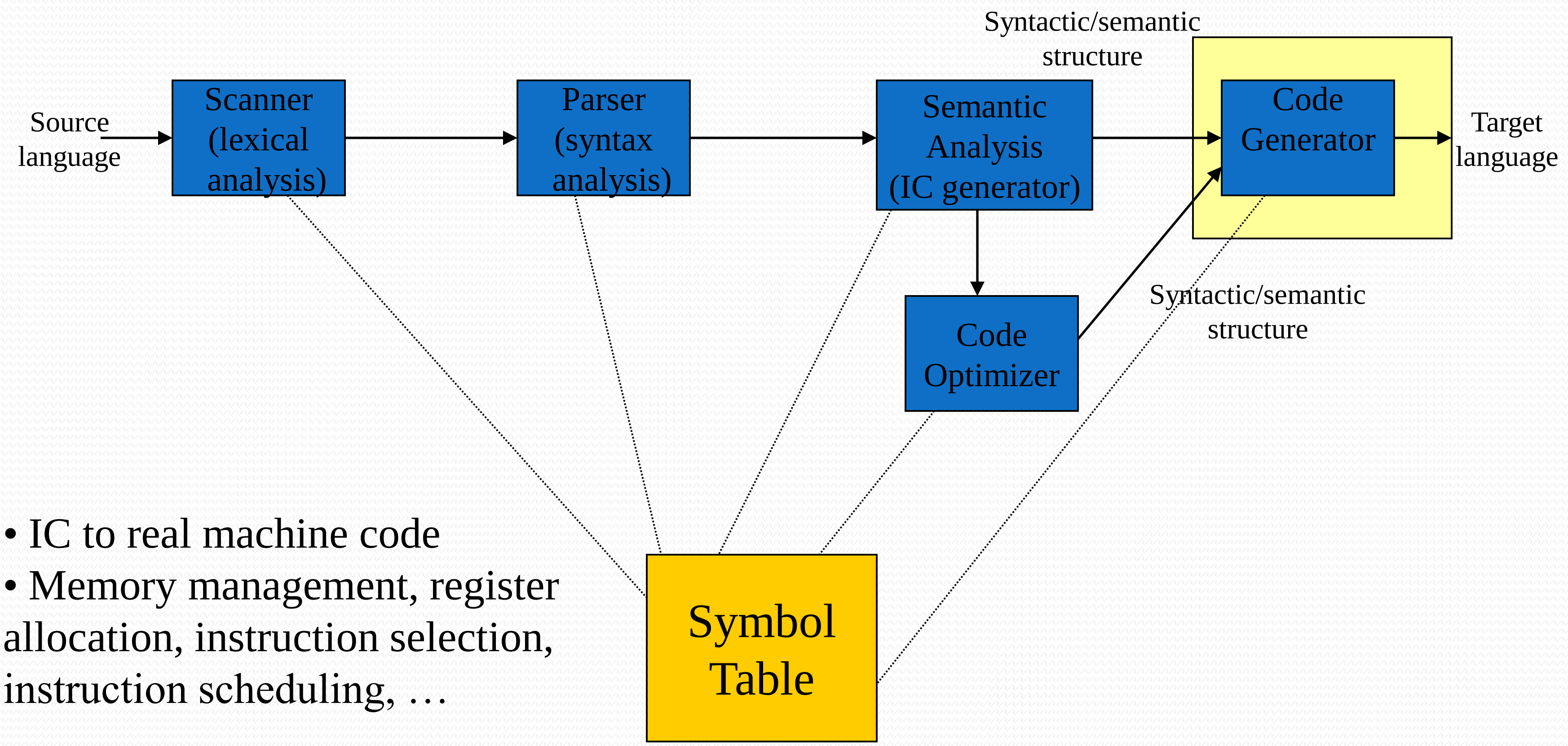
- Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side effects.
- The process of removing unnecessary part of a code is known as code optimization.
- Due to code optimization process it decreases the time and space complexity of the program.
  - *Detection of redundant function calls*
  - *Detection of loop invariants*
  - *Common sub-expression elimination*
  - *Dead code detection and elimination*



```
b   := 0
t1  := a + b
t2  := c * t1
a   := t2
```

```
a  := c*a
b  := 0
```

Code Generation


Source language
Scanner (lexical analysis)
Parser (syntax analysis)
Semantic Analysis (IC generator)
Syntactic/semantic structure
Code Generator
Target language
Code Optimizer
Syntactic/semantic structure
Symbol Table

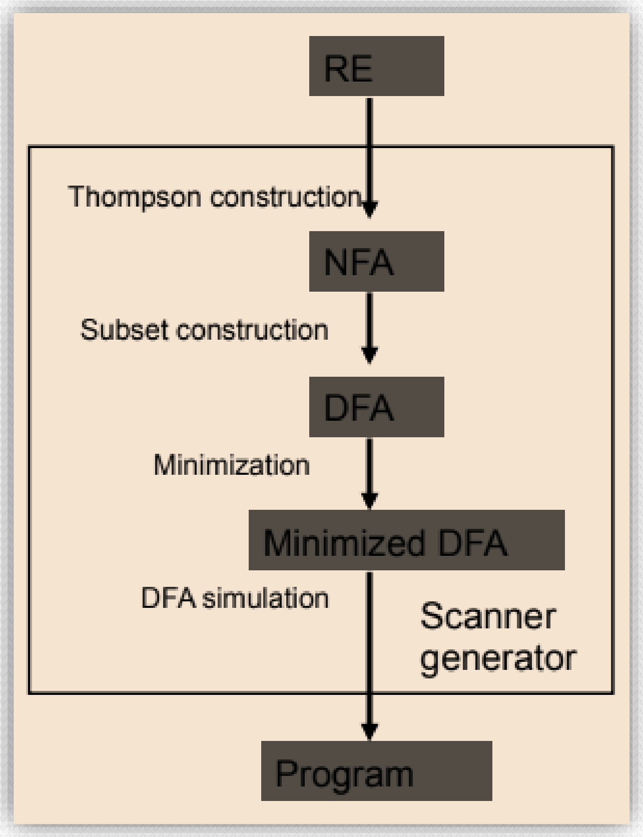
- IC to real machine code
- Memory management, register allocation, instruction selection, instruction scheduling, …

Lexical Analysis with DFA

- The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language.
- It searches for the pattern defined by the language rules.
- Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols.
- The grammar defined by regular expressions is known as regular grammar.
- The language defined by regular grammar is known as regular language.

Lexical Analysis with DFA

- We then design a DFA for the regular expression which follows some pattern so that the tokens can be generated by lexical analyzer.



Lexical Analysis with DFA

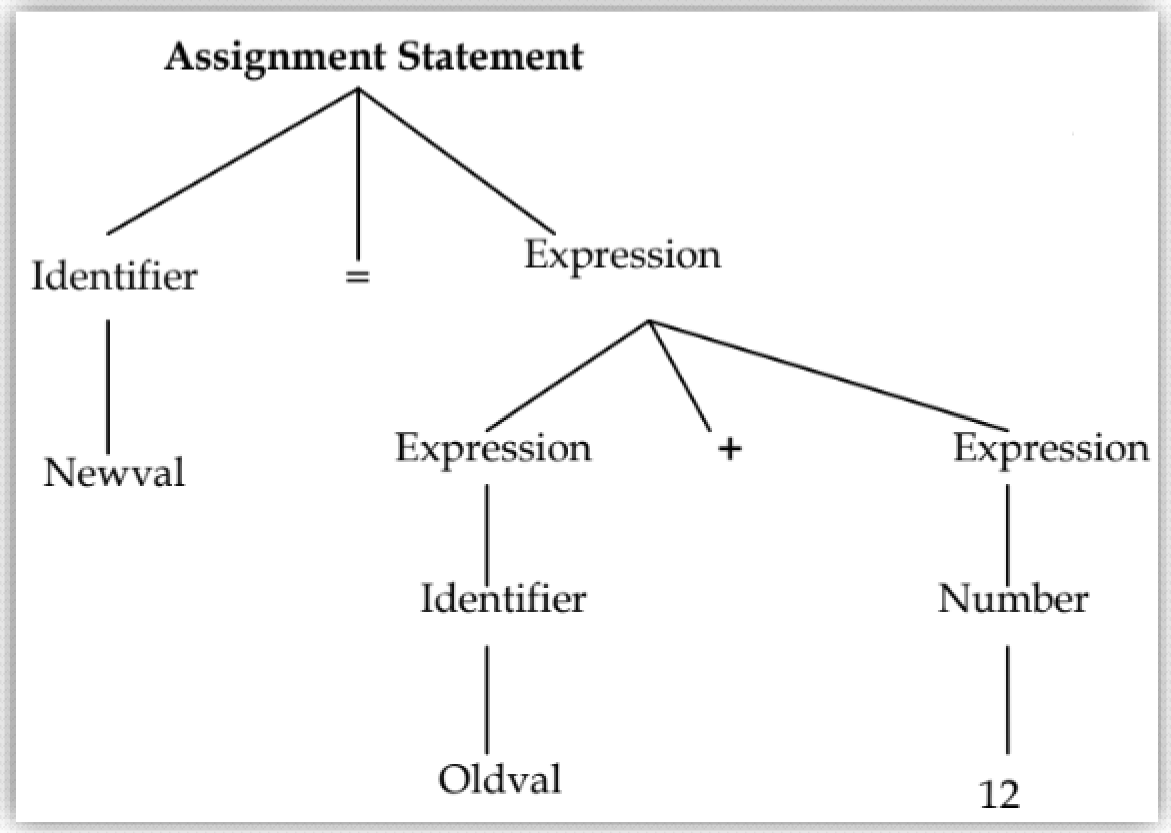
- ***Design a DFA that recognizes the email address***

Syntax Analyzer (Parser)

- A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given source program.
- Its job is to analyze the source program based on the definition of its syntax.
- It works in lock-step with the lexical analyzer and is responsible for creating a parse-tree of the source code.

Syntax Analyzer (Parser)

Example: ***newval* = *oldval* + *12***



Syntax Analyzer (Parser)

- A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams.
- The parser analyzes the source code (token stream) against the production rules to detect any errors in the code.
- The output of this phase is a parse tree.
- The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.
- It has to report any syntax errors if occurs.

Syntax Analyzer (Parser)

- The tasks of parser can be expressed as:
  - *Analyzes the context free syntax*
  - *Generates the parse tree*
  - *Provides the mechanism for context sensitive analysis*
  - *Determine the errors and tries to handle them*

CFG in Syntax Analysis

- Context-Free Grammar (CFG) plays an important role in describing the syntax of programming languages by defining a set of rules for constructing valid strings of symbols.
- Parsing can be done in:
  - ***Top-Down Parsing (Leftmost Derivation)***
  - ***Bottom-Up Parsing (Rightmost Derivation)***

CFG in Syntax Analysis

- ***Top-Down Parsing (Leftmost Derivation)***
  - *Top-down approach, meaning they start from the root of the parse tree and try to match the input string by recursively applying production rules.*
  - *They operate by repeatedly selecting the leftmost non-terminal symbol and replacing it with the corresponding production rule.*
  - *This process continues until the entire input string is parsed or until a mismatch occurs.*

CFG in Syntax Analysis

- ***Bottom-Up Parsing (Rightmost Derivation)***
  - *Bottom-up, meaning they begin parsing by identifying the sequence of terminals and non-terminals that match the production rules in reverse.*
  - *This technique involves shifting input symbols onto a stack until a rule can be reduced.*

Thank You !!!!