

# System Programming and Compiler Construction



## **MODULE 5**

### **COMPILERS : ANALYSIS PHASE**

Prof. Sonal Shroff  
Computer Engineering Department  
TSEC

# Syllabus topics

2

- Introduction to Compilers
- Phases of compilers:
- Lexical Analysis
  - Role of Finite State Automata in Lexical Analysis
  - Design of Lexical analyzer, data structures used .
- Syntax Analysis-
  - Role of Context Free Grammar in Syntax Analysis
  - Types of Parsers:
    - Top down parser- LL(1)
    - Bottom up parser- SR Parser
    - Operator precedence parser
    - SLR
  - Semantic Analysis
  - Syntax directed definitions.

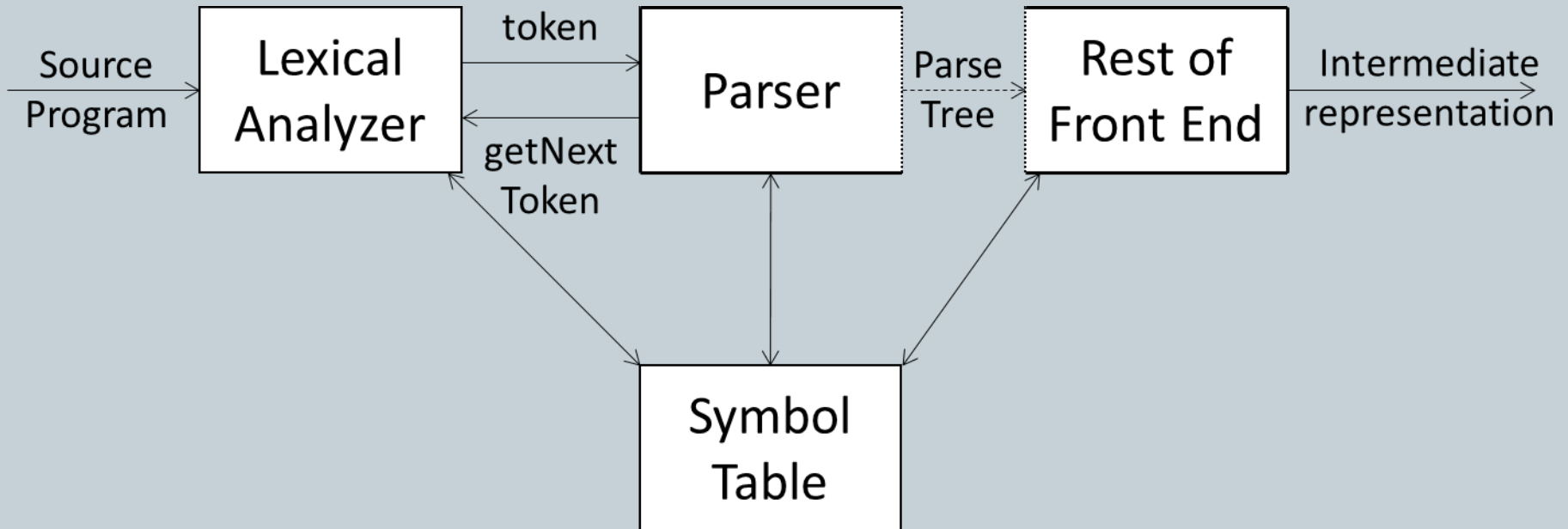
# Syntax Analysis

3

- Syntax analysis is the second phase of the compiler
- It is also called as parsing and it generates parse tree
- Parser: It is the program that takes tokens and grammar (context-free grammar -CFG) as input and validates the input token against the grammar

# Syntax Analysis

4



**Position of Parser in compiler model**

# Syntax Analysis

5

## ➤ Syntax error handling

- **Lexical:** such as misspelling a keyword.
- **Syntactic:** such as an arithmetic expression with unbalanced parentheses.
- **Semantic,** such as an operator applied to an incompatible operand.
- **Logical:** such as an infinitely recursive call

# Syntax Analysis

6

## ➤ Syntax error handling

Goals of Error handler in a parser (simple to state but challenging to realize) :

- Report the presence of errors clearly and accurately
- Recover from each error quickly enough to detect subsequent errors
- Add minimal overhead to the processing of correct programs

# Syntax Analysis

7

## ➤ Error handling strategies

- Panic Mode recovery
- Phrase Level recovery
- Error Productions
- Global Correction

# Syntax Analysis

8

## ➤ Error handling strategies

### ➤ Panic Mode recovery

- Panic mode error recovery is based on the idea of discarding input symbols one at a time until one of the designated set of synchronized tokens is found
- The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source program is clear and unambiguous.
- Advantage of simplicity and does not go into an infinite loop.



# Syntax Analysis

9

## ➤ Error handling strategies

### ➤ Phrase Level recovery

- On discovering error, a parser may perform a local correction on remaining input that allows the parser to continue.
- A typical local correction –
  - Replace comma by semicolon
  - Delete extraneous semicolon
  - Insert a missing semicolon
- Is major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection

# Syntax Analysis

10

## ➤ Error handling strategies

### ➤ Phrase Level recovery

- In phrase level recovery mode each empty entry in the parsing table is filled with a pointer to a specific error routine to take care of that error
- These error routine may be:
  1. Change , insert, or delete input symbol
  2. Issue an appropriate error message
  3. Pop items from the stack

# Syntax Analysis

11

## ➤ Error handling strategies

### ➤ Error productions

- Error production adds rules to the grammar that describes the erroneous syntax .
- This strategy can resolve many , but not all potential errors
- It includes production for common errors and we can augment the grammar for the production rules that generates the erroneous constructs
- A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing
- Since it is almost impossible to know all the errors that can be made by the programmers , this method is not practical

# Syntax Analysis

12

## ➤ Error handling strategies

### ➤ Global correction

- Replace incorrect input with correct input with as few changes as possible.
- This requires expensive techniques that are costly in terms of time and space.

The algorithm states that:

- For a given grammar  $G$  and an incorrect input string  $X$
- Now find a parse tree for a related string  $Y$  with the help of an algorithm such that the number of insertions, deletion and changes of token required to transform  $X$  into  $Y$  are as small as possible

•

# Syntax Analysis

13

## ➤ Context Free Grammars

- Syntax of a language is specified using a notation called – Context Free Grammar (CFG)
- A grammar naturally describes the hierarchical structure of most programming language constructs.
- Example –
  - $\text{statement} \rightarrow \text{if (expression) statement else statement}$   
specifies the structure of this form of conditional statement.

# Syntax Analysis

14

## ➤ Context Free Grammars

- **Terminals** – basic symbols from which strings are formed. Also called as “token name” or “token”.
- **Nonterminal** – syntactic variables that denote sets of strings. They help define the language generated by the grammar. They impose a hierarchical structure on the language.
- **Start symbol** – set of strings denoted by start symbol is the language generated by the grammar.  
productions for the start symbol are listed first

# Syntax Analysis

15

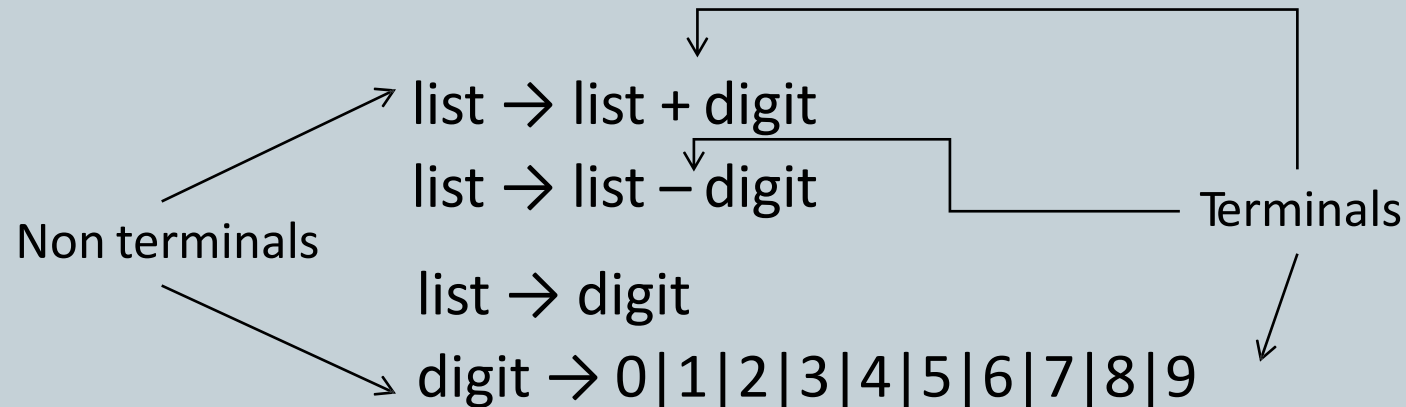
## ➤ Context Free Grammars

- **Productions** – specify manner in which terminals and non-terminals can be combined to form strings.
- Each production consists of –
  - A Nonterminal : head of the production. Defines some strings denoted by the head
  - The symbol “→”
  - A Body or right side : consisting of 0 or more terminals and non-terminals. Describes one way in which strings of the nonterminal at the head can be constructed.

# Syntax Analysis

16

## ➤ Context Free Grammars



- Above grammar can also be written as

$\text{list} \rightarrow \text{list} + \text{digit} \mid \text{list} - \text{digit} \mid \text{digit}$

$\text{digit} \rightarrow 0|1|2|3|4|5|6|7|8|9$



# Syntax Analysis

17

## ➤ Context Free Grammars

- Grammar for simple arithmetic expression

$\text{expr} \rightarrow \text{expr} + \text{term}$

$\text{expr} \rightarrow \text{expr} - \text{term}$

$\text{expr} \rightarrow \text{term}$

$\text{term} \rightarrow \text{term} * \text{factor}$

$\text{term} \rightarrow \text{term} / \text{factor}$

$\text{term} \rightarrow \text{factor}$

$\text{factor} \rightarrow ( \text{expr} )$

$\text{factor} \rightarrow \text{id}$

# Syntax Analysis

18

## ➤ Context Free Grammars

### ➤ Using notational conventions

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow ( E ) \mid \text{id}$$

- Notational conventions tell us that E, T and F are nonterminals, with the start symbol E. The remaining symbols are terminals.

# Syntax Analysis

19

## ➤ Notational conventions

### ➤ These symbols are terminals

- Lowercase letters early in the alphabet , such as a,b,c
- Operator symbols like +, \* and so on
- punctuation symbols like (, ; ...
- The digits 0,1,2,...,9
- Boldface strings like **id**, **if**, each represents a single terminal symbol

# Syntax Analysis

20

## ➤ Notational conventions

### ➤ These symbols are nonterminals

- Uppercase letters early in the alphabet such as A,B,C.
- The letter S, which is usually the start symbol
- Lowercase italic names like *expr*, *stmt*
- Nonterminals for expressions, terms, factors are often represented by E, T and F respectively

# Syntax Analysis

21

## ➤ Notational conventions

- Uppercase letters late in the alphabet such as X, Y, Z represent grammar symbols that is either terminals or nonterminals.
- Lowercase letters late in the alphabet such as u, v...z, represent (possibly empty) strings of terminals
- Lowercase Greek letters  $\alpha$ ,  $\beta$ ,  $\gamma$ , represent (possibly empty) strings of grammar symbols

The generic production can be written as  $A \rightarrow \alpha$

where A is the head and  $\alpha$  is the body

# Syntax Analysis

22

## ➤ Notational conventions

- A set of productions  $A \rightarrow \alpha_1$  ,  $A \rightarrow \alpha_2$  ,...,  $A \rightarrow \alpha_k$  with a common head is called A-productions

written as  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$

- The head of the first production is the start symbol, unless stated otherwise.