System Programming and Compiler Construction

MODULE 5(5)
COMPILERS : ANALYSIS PHASE

Prof. Sonal Shroff
Computer Engineering Department
TSEC

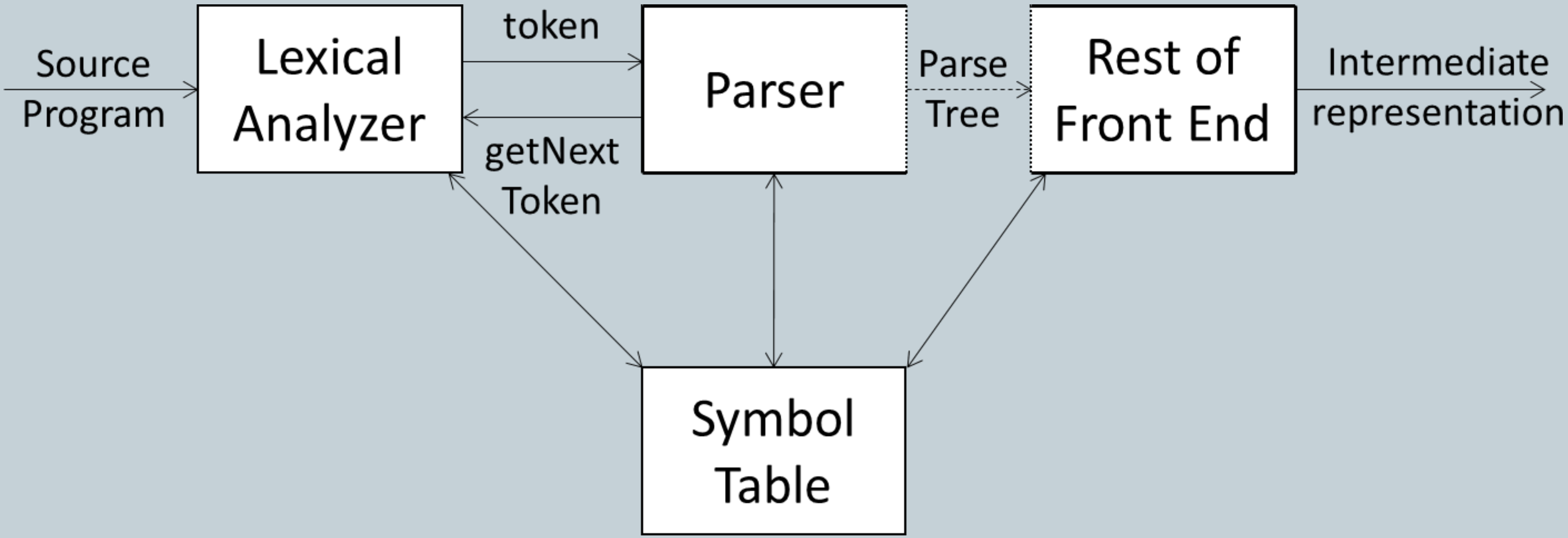
Syllabus topics

- 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

- 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



Position of Parser in compiler model

Syntax Analysis

➢ 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

➢ 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

- Error handling strategies

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

- 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.

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

- 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

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

Error handling strategies

<u>Global correction</u>

- 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

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 –
  - statement → ***if*** (expression) statement ***else*** statement

specifies the structure of this form of conditional statement.

Syntax Analysis

➢ 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

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

Context Free Grammars

list → list + digit

list → list – digit

Terminals

Non terminals

list → digit

digit → 0|1|2|3|4|5|6|7|8|9

- Above grammar can also be written as

list → list + digit | list – digit | digit

digit → 0|1|2|3|4|5|6|7|8|9

Context Free Grammars

- Grammar for simple arithmetic expression

expr → expr + term

expr → expr – term

expr → term

term → term * factor

term → term / factor

term → factor

factor → (expr)

factor → id

- Context Free Grammars

- Using notational conventions

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

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

Syntax Analysis

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

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

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 α, β, γ, represent (possibly empty) strings of grammar symbols

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

where A is the head and α is the body

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 $A \rightarrow \alpha 1 \mid \alpha 2 \mid \ldots \mid \alpha k$

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