Chapter 3

Syntax Analysis

Outline

- Syntax Analysis
- Role of Parser
- Error Handling
- Context-Free Grammars
- Derivations
- Parse Tree
- Ambiguous grammars
- Extended Backus Naur Form
- Top Down Parsing techniques
- Bottom Up Parsing techniques
- JavaCC Parser Generator

Objective

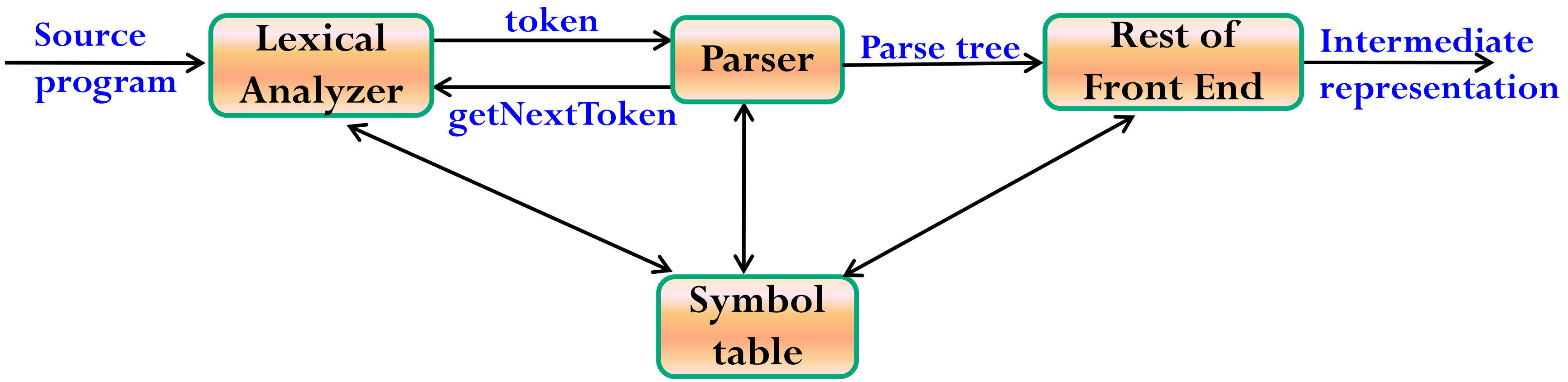
At the end of this chapter students will be able to:

- Understand the basic roles of **Parser(Syntactic Analyzer)**.
- Understand context-Free Grammars(CFGs) and their representation format.
- Understand the different derivation formats: Leftmost derivation, Rightmost derivation and Non-Leftmost, Non-Rightmost derivations
- Be familiar with CFG shorthand techniques.
- Understand **Parse Tree** and its structure.
- Understand ambiguous grammars and how to deal with ambiguity from CFGs.
- Understand the **Extended Backus Naur Form**
- Understand the **JavaCC** Parser Generator and its Structure.

Syntax Analysis

- By design, ***every programming language*** has precise rules that prescribe the syntactic structure of **well-formed programs.**
- The syntax of programming language constructs can be specified by **context-free grammars** or BNF notation.
- The use of CFGs has several advantages over BNF:
  - helps in identifying ambiguities
  - a grammar gives a precise yet easy to understand syntactic specification of a programming language
  - it is possible to have a tool which produces automatically a parser using the grammar
  - a properly designed grammar helps in modifying the parser easily when the **language changes.**

The Role of the Parser



- *Syntax Analyzer* creates the syntactic structure of the given source program.
- This syntactic structure is mostly a *parse tree*.
- Syntax Analyzer is also known as *parser*.
- The syntax of a programming is described by a *context-free grammar (CFG)*. We will use BNF (Backus-Naur Form) notation in the description of CFGs.

Contd…

- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
  - If it satisfies, the parser creates the parse tree of that program.
  - Otherwise the parser gives the error messages.
- A context-free grammar
  - gives a precise syntactic specification of a programming language.
  - the design of the grammar is an initial phase of the design of a compiler.
  - a grammar can be directly converted into a parser by some tools.
- The parser works on stream of tokens.

Contd…

- We categorize the parsers into two groups:

  1. **Top-Down Parser**
     - the parse tree is created top to bottom, starting from the root.
  2. **Bottom-Up Parser**
     - the parse is created bottom to top; starting from the leaves

- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).

- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing
  - LR for bottom-up parsing

Error Handling

Common Programming Errors include:

- Lexical errors, Syntactic errors, Semantic errors and logical Errors

Error handler goals

- 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

Common Error-Recovery Strategies includes:

1. **Panic mode recovery:-** Discard input symbol one at a time until one of designated set of synchronization tokens is found.

2. **Phrase level recovery:-** Replacing a prefix of remaining input by some string that allows the parser to continue.

3. **Error productions:-** Augment the grammar with productions that generate the erroneous constructs

4. **Global correction:-** Choosing minimal sequence of changes to obtain a globally least-cost correction

Context-Free Grammars (CFGs)

- CFG is used as a tool to describe the syntax of a programming language.
- A CFG includes 4 components:

1. A set of terminals ***T***, which are the tokens of the language
   - Terminals are the basic symbols from which strings are formed.
   - The term "token name" is a synonym for "terminal"
2. A set of non-terminals ***N***
   - Non-terminals are syntactic variables that denote sets of strings.
   - The sets of strings denoted by non-terminals help define the language generated by the grammar.
   - Non-terminals impose a hierarchical structure on the language that is key to syntax analysis and translation
3. A set of rewriting rules ***R***.
   - The **left-hand** side (**head**) of each rewriting rule is a **single non-terminal**.
   - The **right-hand** side (**body**) of each rewriting rule is a **string of terminals** and/or **non-terminals**
4. A special non-terminal $S \in N$, which is the start symbol

Contd…

- Just as regular expression generate strings of **characters**, CFG generate strings of **tokens**
- A string of tokens is generated by a CFG in the following way:
  1. The initial input string is the start symbol ***S***
  2. While there are non-terminals left in the string:
     i. Pick any non-terminal in the input string *A*
     ii. Replace a single occurrence of *A* in the string with the right-hand side of any rule that has *A* as the left-hand side
     iii. Repeat 1 and 2 until all elements in the string are terminals

Example: **Terminals** = { id, num, if, then, else, print, =, {, }, ;, (,) }

Non-Terminals = { S, E, B, L }

Rules = (1) S → print(E);

(2) S → while (B) do S

(3) S → { L }

(4) E → id

(5) E → num

(6) B → E > E

(7) L → S

(8) L → SL

Start Symbol = S

Contd…

Example 3: A grammar that defines simple arithmetic expressions:

Terminals = { **id**, **+**, **-**, *****, **/**, **(**, **)** }

Non-Terminals = {**expression**, **term**, **factor** }

Start Symbol = **expression**

Rules = expression → expression + term

→ expression – term

→ term

term → term* factor

→ term/factor

→ factor

factor → (expression)

→ id

Example 4:

1. expression → expression + expression
2. expression → expression – expression
3. expression → expression * expression
4. expression → expression / expression
5. expression → num

expression → expression + expression

→ expression * expression + expression

→ num * expression + expression

→ num * num+ expression

→ num * num+ num

Conventions

1. **These symbols are terminals**:
   - A. Lowercase letters early in the alphabet, such as a, b, c.
   - B. Operator symbols such as +, *, and so on .
   - C. Punctuation symbols such as parentheses , comma, and so on.
   - D. The digits 0, 1, … ,9 .
   - E. Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.

2. **These symbols are non-terminals:**
   - i. Uppercase letters early in the alphabet, such as A, B, C.
   - ii. The letter S, which, when it appears, is usually the start symbol.
   - iii. Lowercase, italic names such as ***expr*** or ***stmt***.
   - iv. Uppercase letters may be used to represent non-terminals for the constructs. **For example:-** non terminals for expressions, terms, and factors are often represented by E, T, and F, respectively.

3. Uppercase letters late in the alphabet , such as X, Y, Z, represent **grammar symbols**; that is , either non-terminals or terminals.

Contd…

4. Lowercase letters late in the alphabet , chiefly u, v, … ,z , represent (possibly empty) strings of terminals.
5. Lowercase Greek letters α, γ, β, for example, represent (possibly empty) strings of grammar symbols.
   - ❖ Thus, a generic production can be written as $A \rightarrow \alpha$, where A is the head and α the body.
6. A set of productions $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, $A \rightarrow \alpha_3, \ldots,$ $A \rightarrow \alpha_k$ with a common head A (call them A-productions), may be written $A \rightarrow \alpha_1 \mid A \rightarrow \alpha_2 \mid A \rightarrow \alpha_3 \mid \ldots \mid A \rightarrow \alpha_k$.
   - ➢ Call $\alpha_1, \alpha_2, {}_3, \ldots, \alpha_k$ the alternatives for A
7. Unless stated otherwise, the head of the first production is the start symbol.

Example:- Using these conventions , the grammar of Example 4 of slide # 9 can be rewritten concisely as:

$E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T*F \mid T/F \mid F$

$F \rightarrow (E) \mid id$

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

Derivations

- A derivation is a description of how a string is generated from the start symbol of a grammar.
    1. A **leftmost derivation** always picks the leftmost non-terminal to replace (see slide 13)
    2. A **rightmost derivation** always picks the rightmost non-terminal to replace(see slide 14)
- For example: Use the CFG below to generate *print (id);*

Terminals = { id, num, if, then, else, print, =, {, }, ;, (,) }

Non-Terminals = { S, E, B, L }

Rules = (1) S $\rightarrow$ print(E);

(2) S $\rightarrow$ while (B) do S

(3) S $\rightarrow$ { L }

(4) E $\rightarrow$ id

(5) E $\rightarrow$ num

(6) B $\rightarrow$ E > E

(7) L $\rightarrow$ S

(8) L $\rightarrow$ SL

Start Symbol = S

Leftmost Derivations

- A string of terminals and non-terminals α that can be derived from the initial symbol of the grammar is called a ***sentential form***
- Thus the strings **"{ S L }", "while(id>E) do S", and print(E>id)"** of the above example re all sentential forms
- A derivation is "leftmost" if, at each step in the derivation, the leftmost non-terminal is selected to replace
  - All of the above examples are leftmost derivations
- A sentential form that occurs in a leftmost derivation is called a **left-sentential form**

Example 1: We can use leftmost derivations to generate **while(id > num) do print(id);** from this CFG as follows:

S → while(B) do S

→ while(E>E) do S

→ while(id>E) do S

→ while(id>num) do S

→ while(id>num) do print(E);

→ while(id>num) do print(id);

Example 2: We also can generate { **print(id); print(num);** } from the CFG as follows:

S → { L }

→ { S L }

→ { print(E); L }

→ { print(id); L }

→ { print(id); S }

→ { print(id); print(E); }

→ { print(id); print(num); }

Rightmost Derivations

- Is a derivation technique that chooses the rightmost non-terminal to replace

Example 1: To generate **while(num > num) do print(id);**

S → while(B) do S

→ while(B) do print(E);

→ while(B) do print(id);

→ while(E>E) do print(id);

→ while(E>num) do print(id);

→ while(num>num) do print(id);

Example 2: Try to derivate { print(num); print(id); } from S

S → { L }

→ { S L }

→ { S S }

→ { S print(E); }

→ { S print(id); }

→ { print(E); print(id); }

→ { print(num); print(id); }

CFG Shorthand

- We can combine two rules of the form $S \rightarrow \alpha$ and $S \rightarrow \beta$ to get the single rule $S \rightarrow \alpha \mid \beta$

Example:

Terminals = { id, num, if, then, else, print, =, {, }, ;, (,) }

Non-Terminals = { S, E, B, L }

Rules = S → print(E); | while (B) do S | { L }

E → id | num

B → E > E

L → S | SL

Start Symbol = S

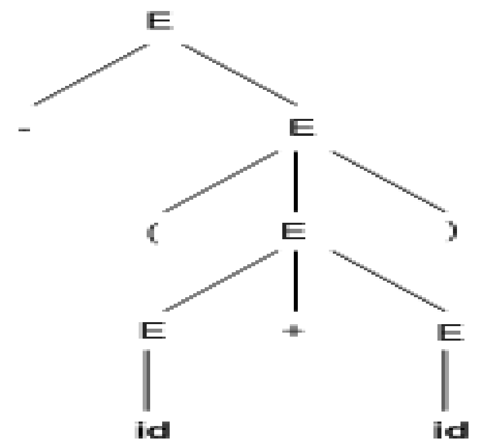
Parse Trees

- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non-terminals .
  - Each interior node of a parse tree represents the application of a production.
  - The interior node is labeled with the **nonterminal A** in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this **A** was replaced during the derivation .
- We start with the initial symbol S of the grammar as the root of the tree
  - The children of the root are the symbols that were used to rewrite the initial symbol in the derivation
  - The internal nodes of the parse tree are non-terminals
  - The children of each internal node N are the symbols on the right-hand side of a rule that has N as the left-hand side (e.g. B $\rightarrow$ E > E where E > E is the right-hand side and B is the left-hand side of the rule)
- Terminals are leaves of the tree.

Examples

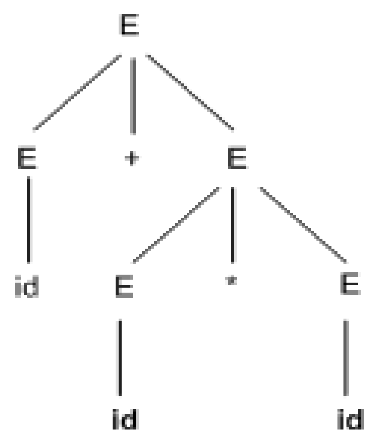
Example 1: -(**id**+**id**)

E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)

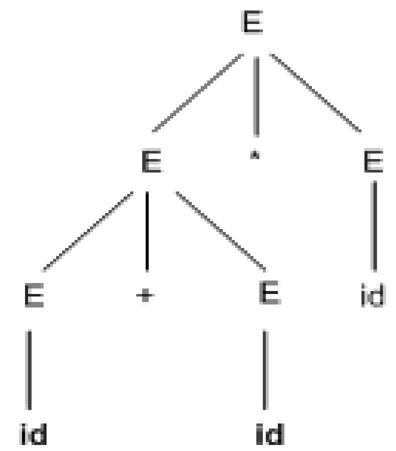


Example 2: (**id**+**id*id**)

E => E+E => E+E*E =>(E+id*E) => (E+id*id)=>(**id**+**id*id**)



a)



b)

Ambiguous Grammars

- A grammar is **ambiguous** if there is **at least one string derivable from the grammar** that has more than one different parse tree, or more than one leftmost derivation, or more than one rightmost derivation
  - Example 2 of slide 18 has two parse trees(parse tree a and b) that are ambiguous grammars.
- Ambiguous grammars are **bad**, because the parse trees don't tell us the exact meaning of the string.
  - For example, in Example 2 of the previous slide, in Fig a. the string means **id*(id+id),** but in Fig. b, the string means **(id*id)+id**. This is why we call it "**ambiguous**".

We need to change the grammar to fix this problem. How? We may rewrite the grammar as follows:

Terminals = { **id**, **+**, **-**, *****, **/**, **(**, **)** }

Non-Terminals = {**E**, **T**, **F** }

Start Symbol = **E**

Rules = $E \rightarrow E + T$

$E \rightarrow E - T$

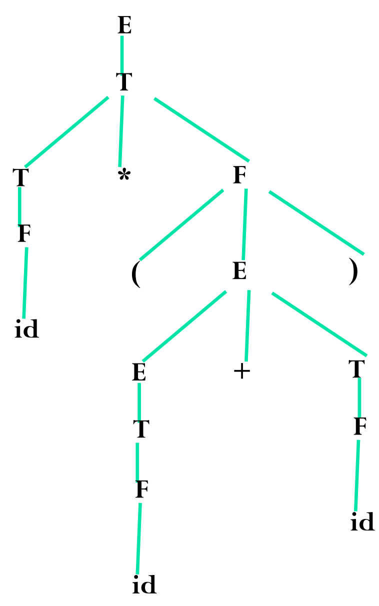
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T \ / \ F$

$F \rightarrow id$

$F \rightarrow (E)$



A parse tree for **id*id(id+id)**

Contd…

We need to make sure that all additions appear higher in the tree than multiplications (Why?)

How can we do this?

- Once we replace an E with E*E using single rule 4, we don't want to rewrite any of the **E**s we've just created using rule 2, since that would place an addition (+) lower in the tree than a multiplication (*)
- Let's create a new **non-terminal T** for multiplication and division
- T will generate strings of id's multiplied or divided together, with no additions or subtractions.
- Then we can modify E to generate strings of T's added together
- This modified grammar is shown at slide no. 19.
- However, this grammar is still **ambiguous**. It is impossible to generate a parse tree from slide no. 19 that has *** higher than + in the tree**

Contd…

- Consider the string id+id+id, which has two parse trees, as shown at example 2 of slide 18:

  id+id+id = (id+id)+id or

  = id+(id+id) are all ok

  id-id-id = (id-id)-id

  != id-(id-id) but this is wrong

- We would like addition and subtraction to have leftmost association as above
  - In other words, we need to make sure that the **right sub-tree** of an addition or subtraction is not another addition or subtraction
- We modified the parse tree of example 2 of slide 18 by the CFG and parse tree shown at slide no. 19 to generate an unambiguous CFG and parse tree.

Extended Backus Naur Form(EBNF)

- Another term for a CFG is a Backus Naur Form (BNF).
- There is an extension to BNF notation, called Extended Backus Naur Form, or EBNF
- EBNF rules allow us to mix and match **CFG** notation and **regular expression** notation in the right-hand side of CFG rules
- For example, consider the following CFG, which describes simpleJava statement blocks and stylized simpleJava print statements:

1. S → { B }
2. S → print(id)
3. B → S ; C
4. C → S ; C
5. C → ε

- Rules 3, 4, and 5 in the above grammar describe a series of one or more statements S, terminated by semicolons

We could express the same language using an EBNF as follows:

1. S → { B }
2. S → print"("id")"
3. B → (S;)+

Note

- In Rule 2, when we want a **parenthesis** to appear in EBNF, we need to surround it with **quotation marks**.
- But in Rule 3, the pair of parenthesis is for the + symbol, not belongs to the language.

Exercise

1. Consider the following grammar

Terminals = { a, b }

Non-Terminals = {**S**, **T**, **F** }

Start Symbol = **S**

Rules = S⟶TF

T⟶T T T

T⟶ a

F⟶ aFb

F⟶ b

Which of the following strings are derivable from the grammar? Give the parse tree for derivable strings?

i. ab

ii. aabb

iii. aba

iv. aaabb

v. aaaabb

vi. aabbb

2. Show that the following CFGs are ambiguous by giving two parse trees for the same string?

2.1) **Terminals** = { a, b }

Non-Terminals = {**S**, **T**}

Start Symbol = **S**

Rules = S⟶ STS

S⟶ b

T⟶ aT

T⟶ ε

2.2) **Terminals** = { if, then, else, print, id }

Non-Terminals = {**S**, **T**}

Start Symbol = **S**

Rules = S⟶ if id then S T

S⟶ print id

T⟶ else S

T⟶ ε

Contd…

3. Construct a CFG for each of the following:

a. All integers with sign (Example: +3, -3)

b. The set of all strings over { **(**, **)**, **[**, **]**} which form balanced parenthesis. That is, (). ()(), ((()())()), [()()] and ([()[]()]) are in the language but)(,][, (() and ([are not.

c. The set of all string over {**num**, **+**, **-**, *****, **/**}which are legal binary post-fix expressions. Thus numnum+, num num num + *, num num – num * are all in the language, while num*, num*num and num num num – are not in the language.

d. Are your CFGs in a, b and c ambiguous?