COMPILER CONSTRUCTION
CS F363

Syntax Analysis

BITS Pilani
Dubai Campus

Dr. R. Elakkiya, AP/CS

Text Book

- ✓Aho, Lam, Sethi and Ullman, “Compilers-Principles, Techniques & Tools”, Pearson/Addison-Wesley, Second Edition, 2013.

innovate achieve lead

BITS Pilani
Dubai Campus

Text Book Reading:

Chapter 4

Section 4.1 to 4.9

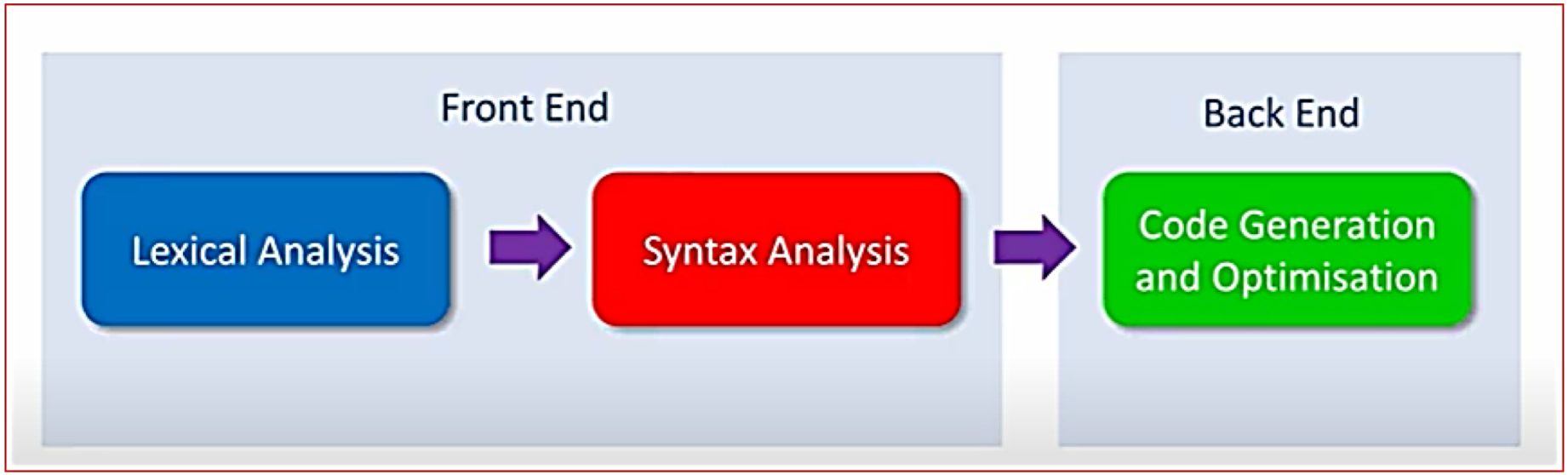
Overview

- ✓Syntax Analysis
  - ✓Introduction
  - ✓Context Free Grammars
  - ✓Writing a Grammar
  - ✓Top-Down Parsing
  - ✓Bottom-Up Parsing
  - ✓Simple LR
  - ✓Powerful LR
  - ✓Using Ambiguous Grammars
  - ✓Parser Generators

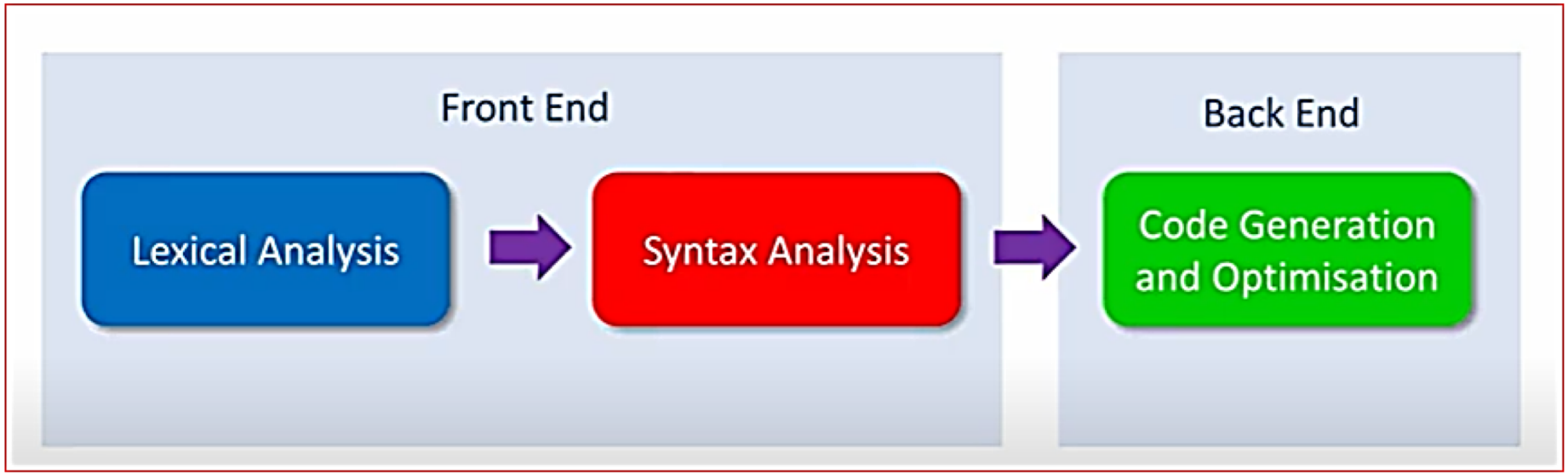
Overview

- ✓Syntax Analysis
  - ✓Introduction
  - ✓Context Free Grammars
  - ✓Writing a Grammar
  - ✓Top-Down Parsing
  - ✓Bottom-Up Parsing
  - ✓LR and RR Grammars
  - ✓Using Ambiguous Grammars
  - ✓Parser Generators

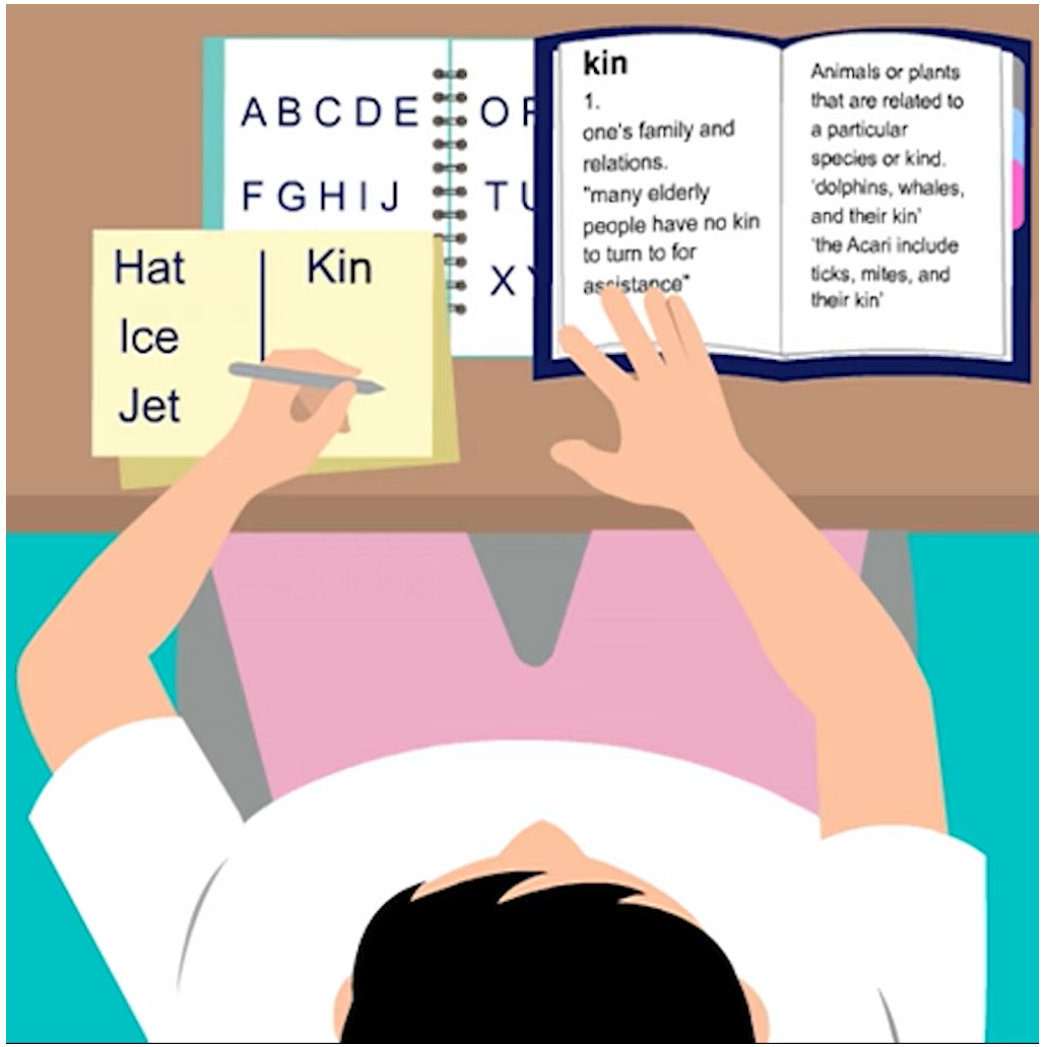
Syntax Analysis



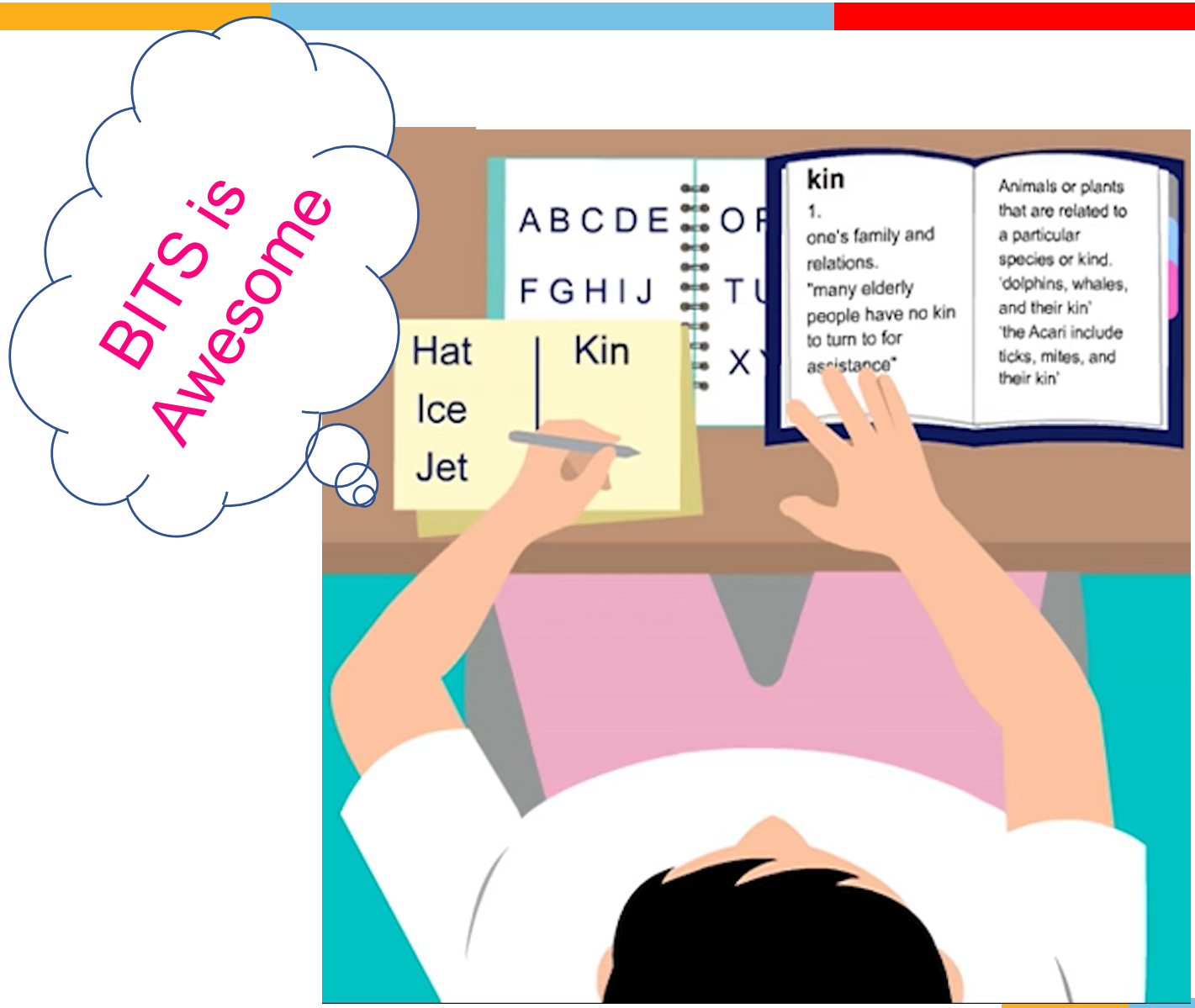
Syntax Analysis



Understanding of Syntax Analsis

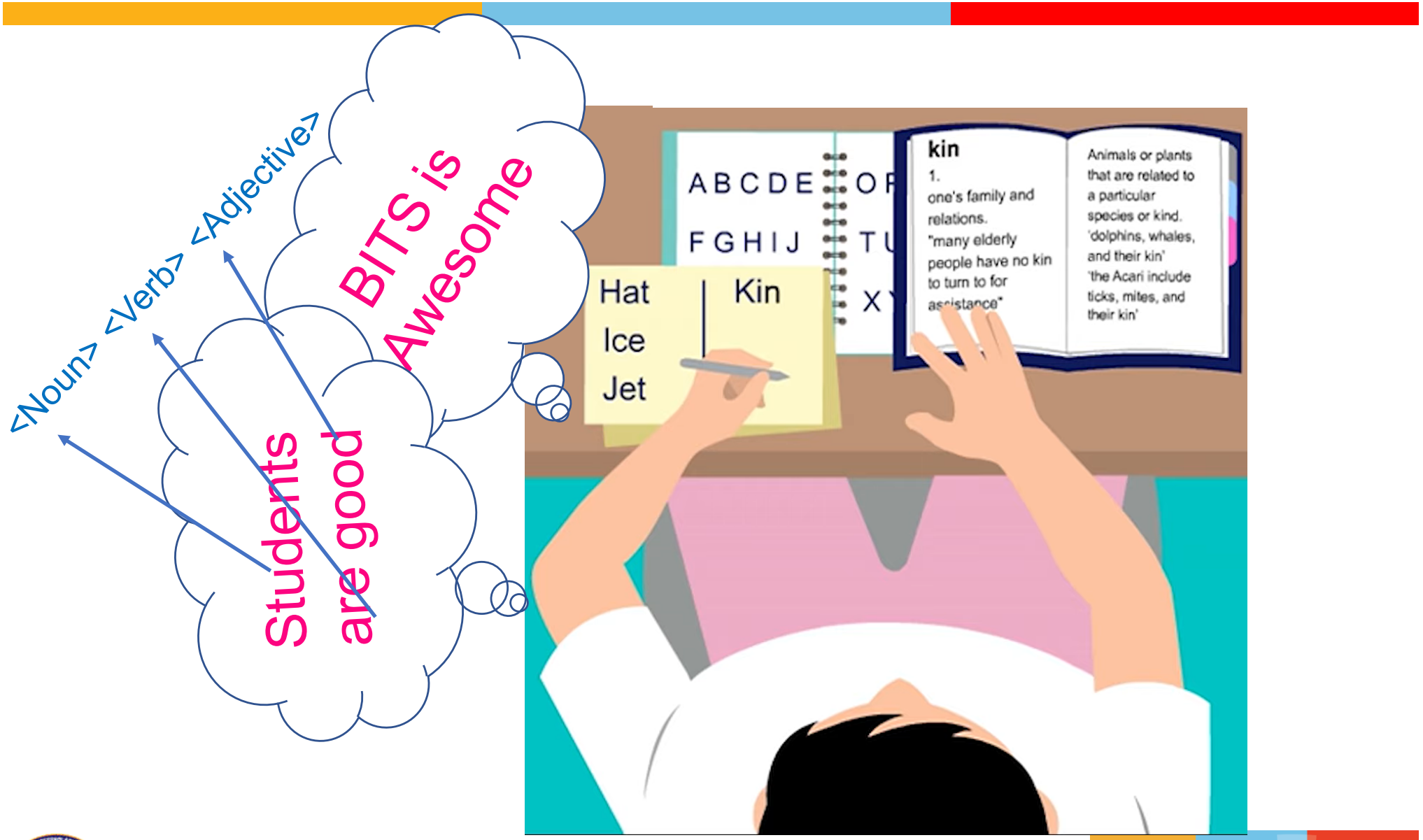


Understanding of Syntax Analsis



Understanding of Syntax Analsis

Understanding of Syntax Analsis



Introduction

Introduction

✓BITS is Awesome

<sentence>

<sentence>: BITS is Awesome

| | |

<Noun> <verb> <Adjective>

| | |

<noun> <Verb Phrase>

<sentence>

Introduction

<if_statement> ::= If <condition> Then <statement>

| If <condition> Then <statements> End If

| If <condition> Then <statements> Else <statements> End If

Introduction

<if_statement> ::= If <condition> Then <statement>

| If <condition> Then <statements> End If

| If <condition> Then <statements> Else <statements> End If

```
If Temperature <= 0 Then
    Description = "Freezing"
End If
```

Introduction

<if_statement> ::= If <condition> Then <statement>
| If <condition> Then <statements> End If
| If <condition> Then <statements> Else <statements> End If

<condition> ::= <expression> <relational_operator> <expression>

<relational_operator> ::= < | > | <= | >= | =

<expression> ::= <expression> + <term> | <expression> - <term> | <term>

<term> ::= <term> * <factor> | <term> / <factor> | <factor>

Introduction

<factor> ::= <identifier> | <number> | (<expression>) | <string_literal>

<statements> ::= <statement> <statements>

<statement> ::= <assignment> | <if_statement> | <for_statement> …

<assignment> ::= <identifier> = <expression>

<identifier> ::= <letter> | <identifier><letter> | <identifier><digit>

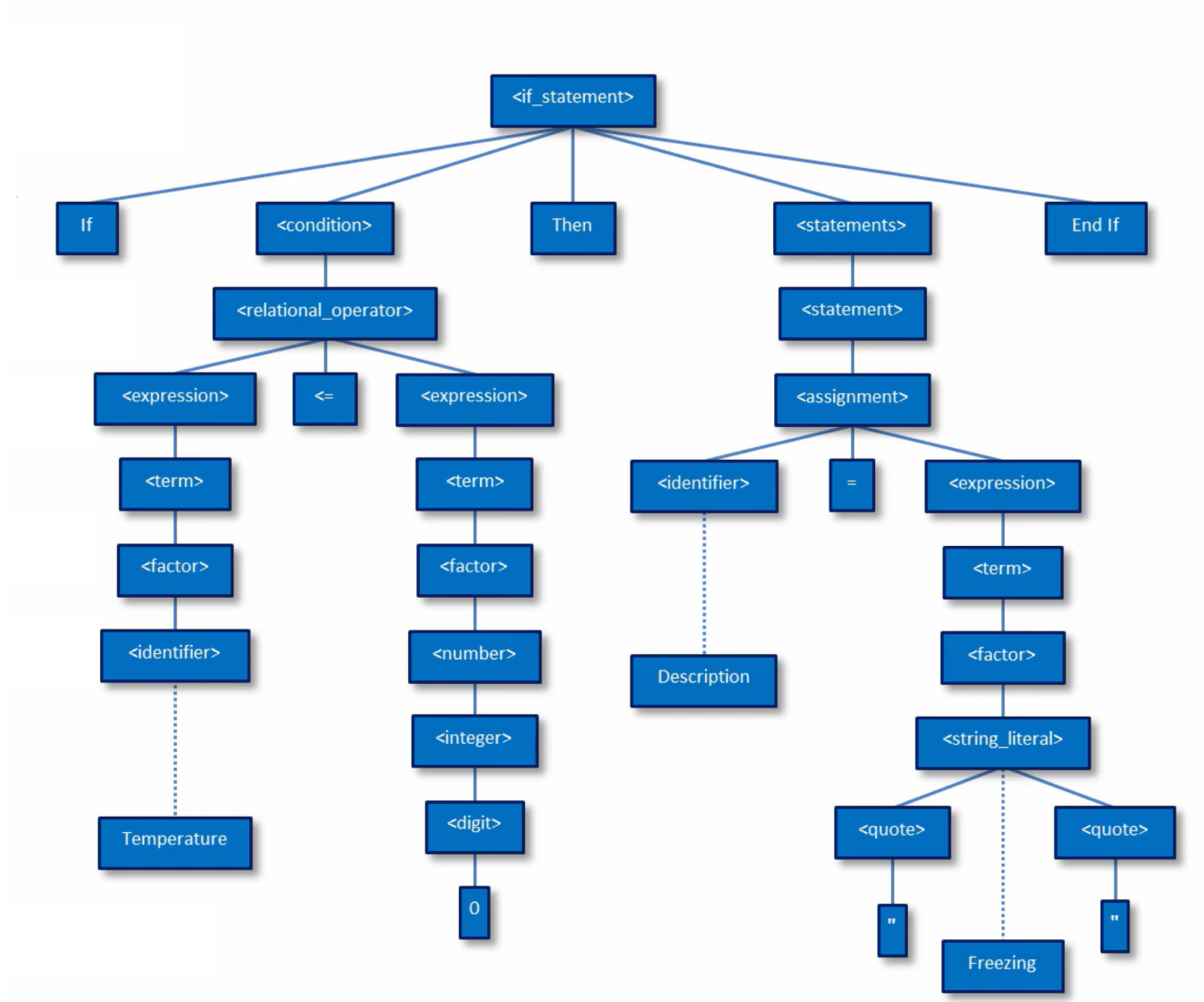
<letter> ::= a | b | c | d | … | A | B | C | D | …

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

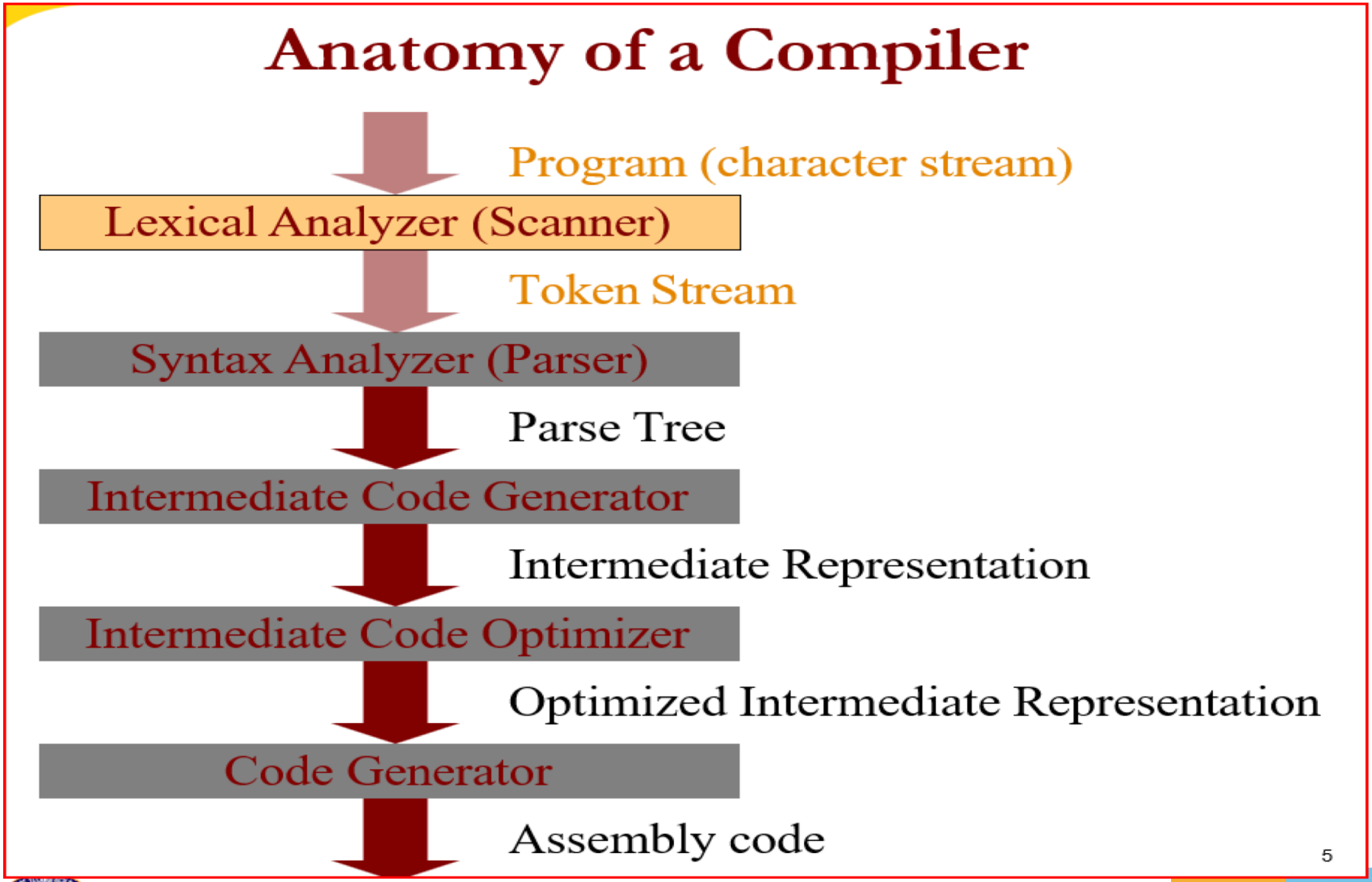
<number> ::= <integer> | <integer>.<integer>

<integer> ::= <digit> | <integer><digit>

Introduction



Overall Compiler Stages



Position of Syntax Analyser

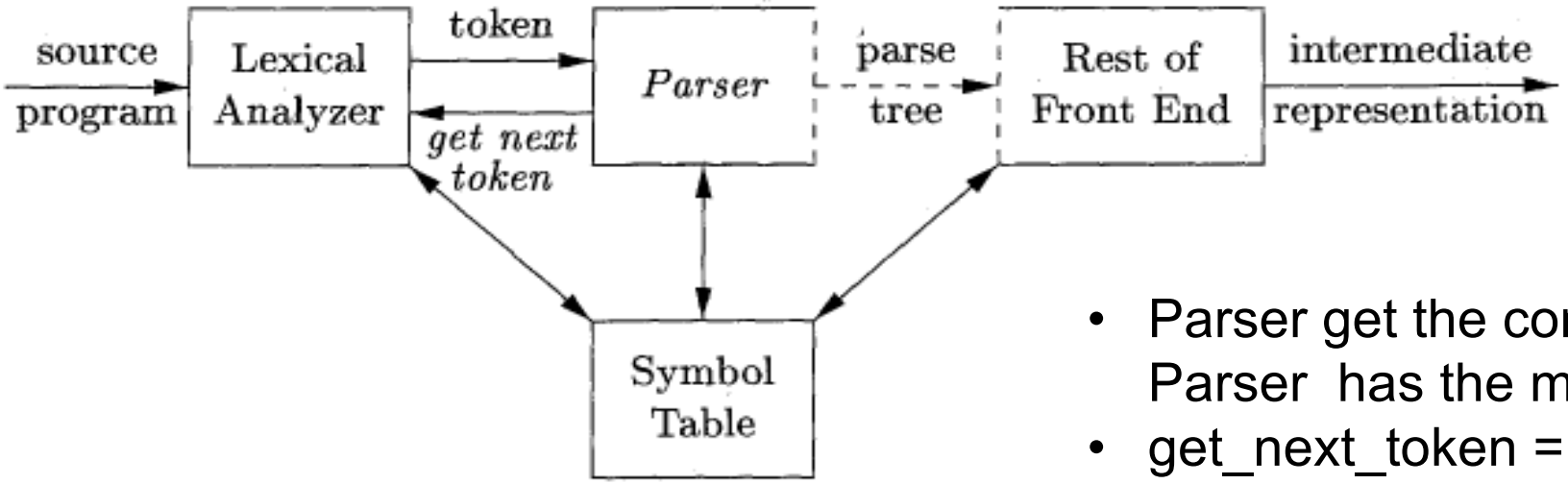


Figure 4.1: Position of parser in compiler model

- Parser get the control first . Ie. Parser has the main()
- get_next_token = getToken() in hand coded parsed yylex() in lex, yacc combination

- ✓ What ?
  - ✓ Lexical Analyzer Creates Tokens out of a Text Stream
- ✓ How :
  - ✓ Tokens are defined using Regular Expressions

Syntax Analyser

Program (character stream)

a = b + c * d

Lexical Analyzer (Scanner)

lex.yy.c ← lex ← a.l

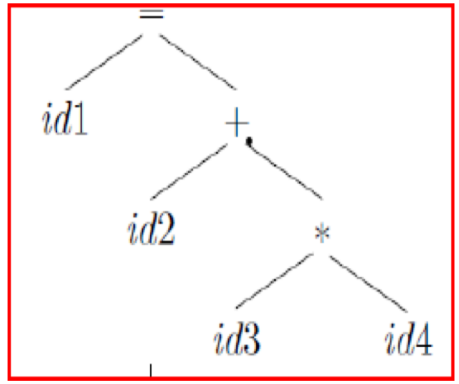
Token Stream

<id,a> <=> <id,b> <+> <id,c> <*> <id,d>

Syntax Analyzer (Parser)

?? ← ?? ← ??

Parse Tree



Intermediate Code Generator

Intermediate Representation

Intermediate Code Optimizer

Optimized Intermediate Representation

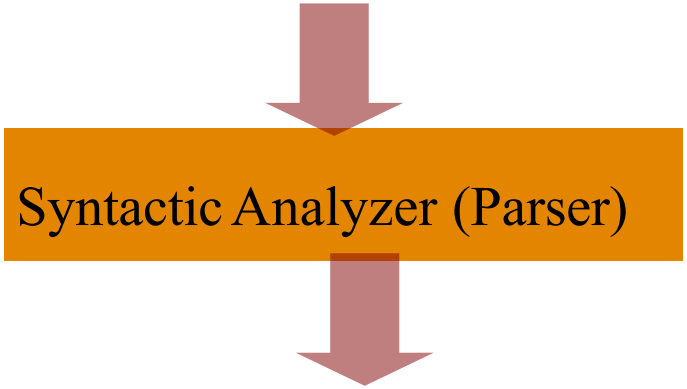
Code Generator

Assembly code

Syntax Analyser Eg.

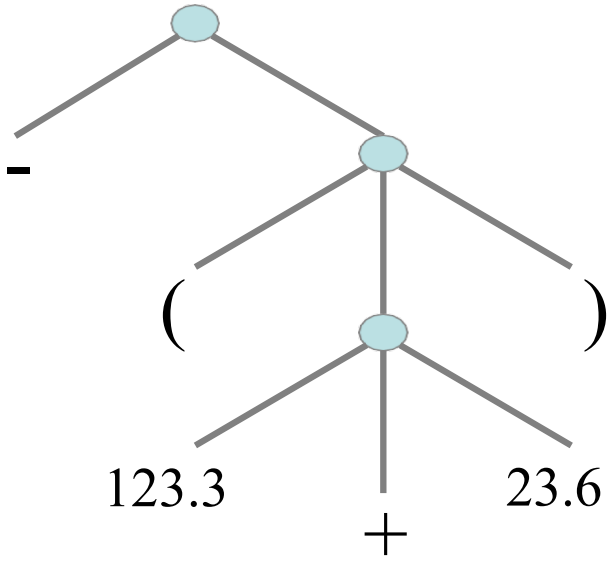
Input: - (123.3 + 23.6)

Token Stream: T_MINUS T_LP T_NUM(123.3) T_PLUS T_NUM(123.3) T_RP



?? ← ?? ← ??

Parse Tree:



Successful build of a parse tree indicates that the input is syntactically correct, which is prereq for final code gen

Lexical Analysis vs Syntax Analysis

Compiler Phase	Input	Output	Compact Description	Automatic Code Gen
Lexer	String of characters	String of Tokens	Regular Expression	Yes (Eg.Using lex -> lex.yy.c)
Parser	String of tokens	Parse tree	Reg Expr ??	parser-generator ??

- ✓Regular Expression recognizes (matches) with valid tokens.
  - ✓Describe the valid tokens.
- ✓Some mechanism to
  - ✓recognizes/match with a string of valid tokens.
  - ✓Describe the valid string of tokens
  - ✓Syntactically correct string of tokens.

Validate a programming Language construct. Eg. Arithmetic Expr if/while/functions etc

Syntax Analysis

- ✓Syntax Analysis = Successful generation of a parse tree
- ✓Look at syntax analysis as a process to
  - ✓Check whether the input string belongs to a language.
  - ✓ Eg. “2 + 3 * 5”, -(-3 + x) etc. are legal (syntactically correct) arithmetic expr..
  - ✓ “ 2 3 * 5”, -(-3 + x etc. are not legal (syntactically incorrect)arithmetic expr
- ✓Can Regular expression be used to specify/recognize ?
  - ✓All Syntactically correct arithmetic expression
  - ✓All programming lang constructs
  - ✓eg. compound/nested statements, while statements etc.
- ✓Reflect on what is its significance from a compiler perspective

Can RE be used for syntax analysis?

- ✓Eg. Programming language constructs

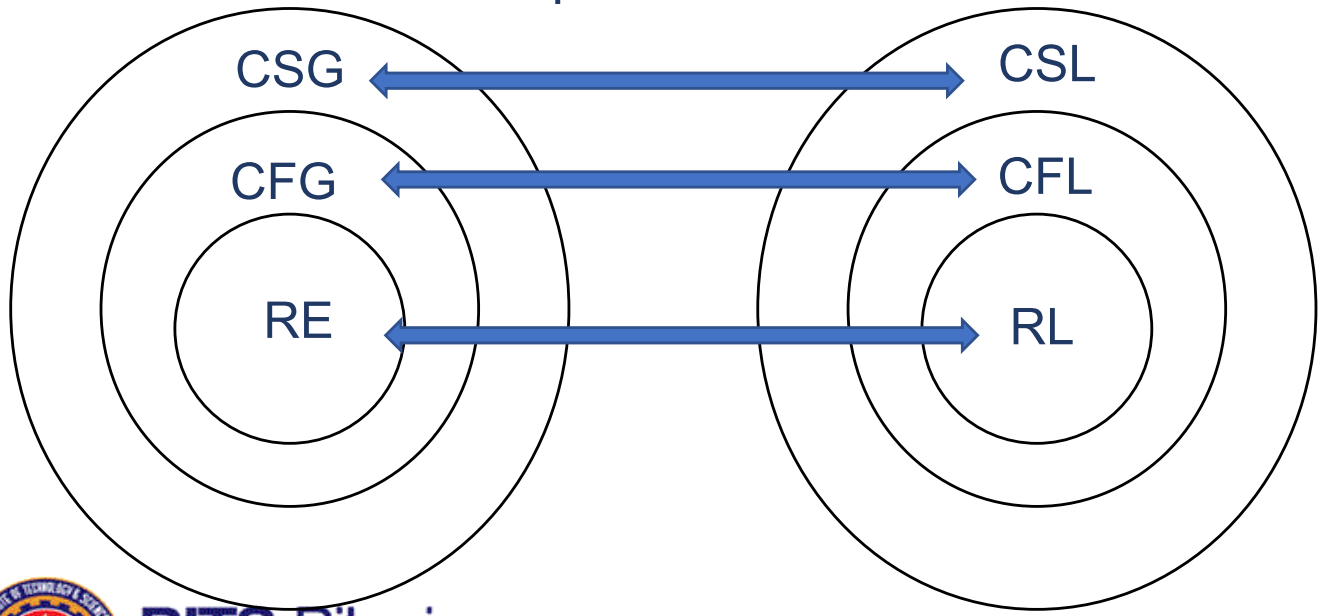
```
(a + b) * (c + d * e)

   if ( a > b) {
      x = x + y;
   } else {
      if ( c > d ) {
         x = x - y;
         c = c * d;
       }
   }
```

- ✓Nested programing Language constructs
  - ✓ equivalent to Balanced parenthesis problem

RE cannot be used syntax analysis.

- ✓Regular Expression cannot be used.. why?
- ✓Because of the nested structure..
  - ✓The order of unbounded nesting needs be kept track of.
  - ✓RE ⇔ NFA ⇔DFA (Finite State Machine)
    - ✓Cannot keep track of unbounded nested structure.
    - ✓Cannot keep track of count



RE/RL = Reg Expr/Lang

CFG/CFL = Context Free Grammar/Lang

CSG/CSL = Context Sensitive Grammar/ Lang

Context Free Grammar & PDA

- ✓Solution: Push down Automata (PDA)
- ✓RE is a compact way to represent DFA (use to recognize RL)
  - ✓ Automatic conversion from RE => DFA (lex a.l -> lex.yy.c)

- ✓How to represent PDA (instead of drawing a state diagram)
  - ✓ Grammar (Context Free)
  - ✓ Natural way to describe the recursive structure of programming language constructs.
  - ✓ is a compact way to represent PDA
  - ✓ Hand-code a parser (from Grammar)
  - ✓ Automatic conversion from CFG => PDA (`yacc a.y -> y.tab.c`)

Syntax Analyzer

Program
(character stream)

a = b + c * d

Lexical Analyzer (Scanner)

lex.yy.c ▸ lex ← a.l

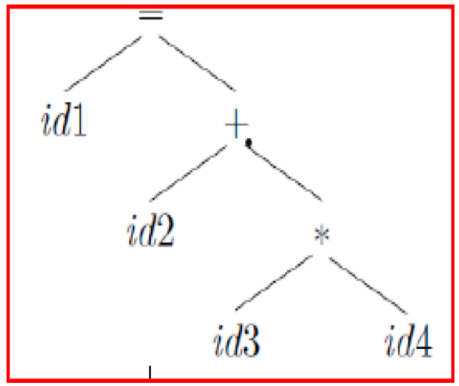
Token Stream

<id,a> <=> <id,b> <+> <id,c> <*> <id,d>

Syntax Analyzer (Parser)

y.tab.c ← yacc ← a.y

Parse Tree



Intermediate Code Generator

Intermediate
Representation

Intermediate Code Optimizer

Optimized
Intermediate Representation

Code Generator

Assembly code

yacc => Yet another compiler compiler

Syntax Analysis

✓BITS is Awesome

<sentence>

<sentence>: BITS is Awesome

| | |

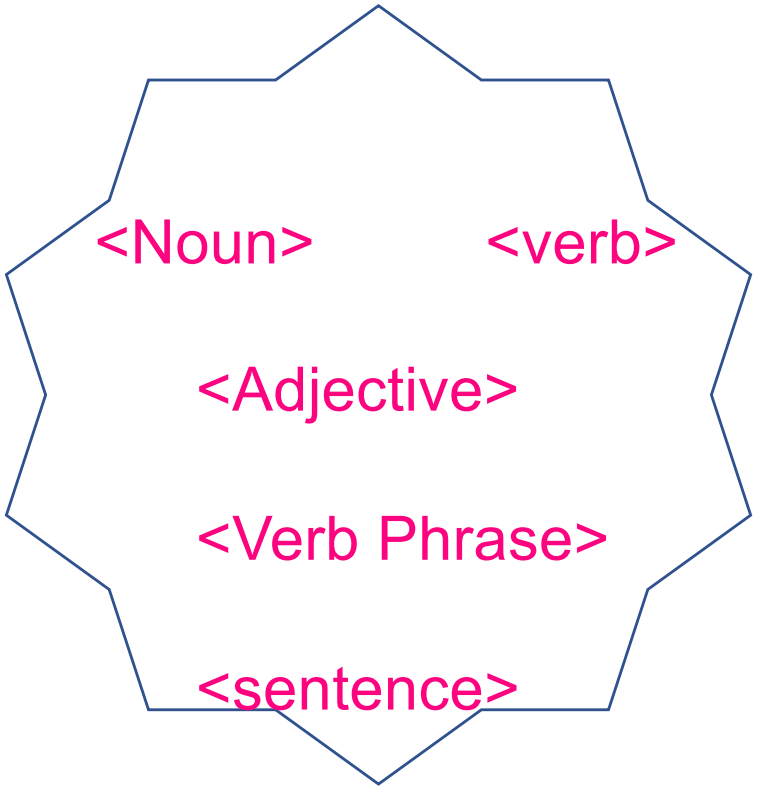
<Noun> <verb> <Adjective>

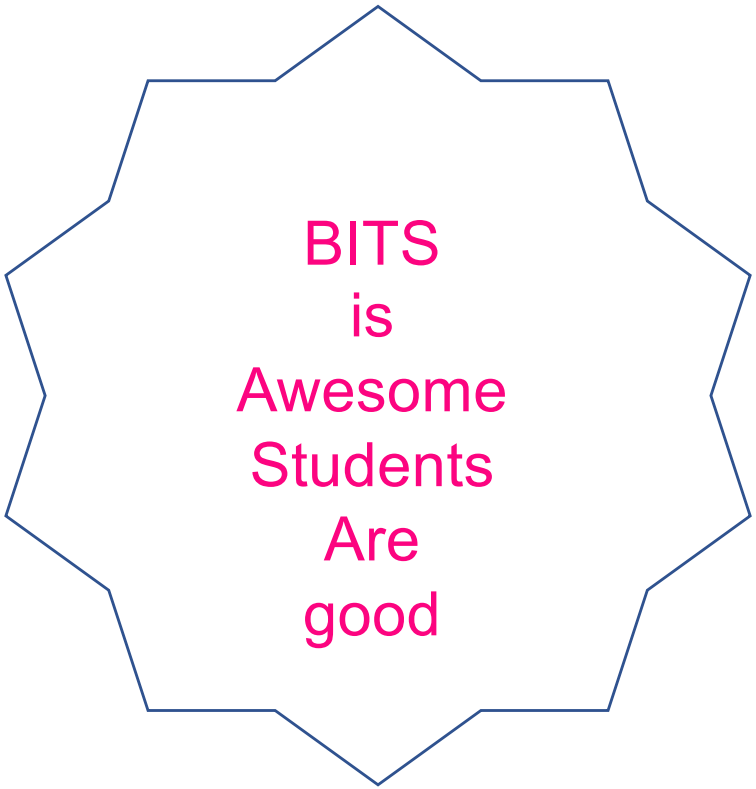
|

<noun> <Verb Phrase>

<sentence>

Grammar





Variables or Non-Terminals

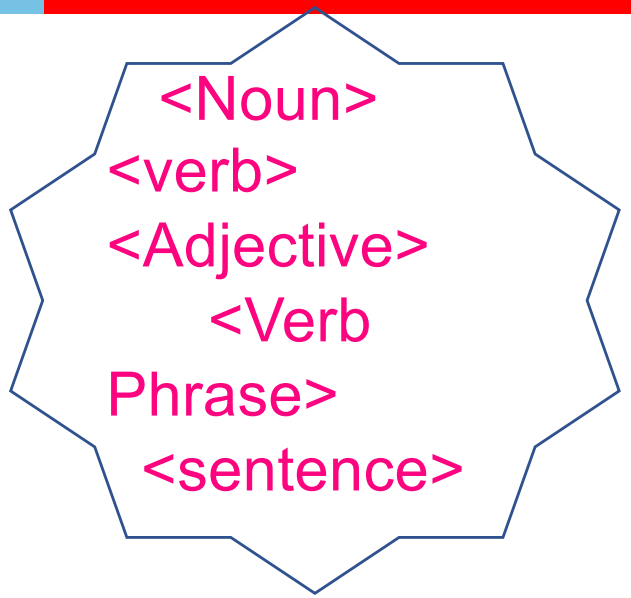
Constants or Terminals

Grammar

Variables or Non-Terminals	<Noun>	<verb>	<Adjective>
Constants or Terminals	BITS	is	Awesome

Grammar

<s> → <Noun> <Verb Phrase>
<Verb Phrase> → <Verb> <Adjective>
<Noun> → BITS | Students
<Verb> → is | are
<Adjective> → awesome | good



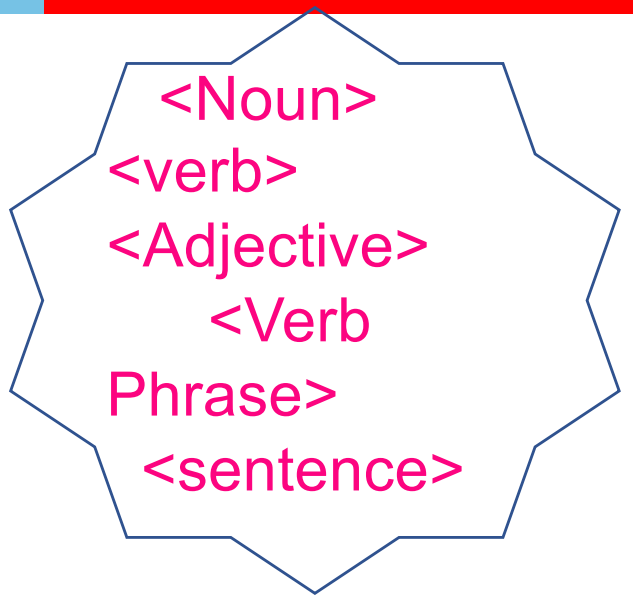
Variables or Non-Terminals

Constants or Terminals

Grammar

Formal Grammar: (N,T,P,S)

<s> → <Noun> <Verb Phrase>
<Verb Phrase> → <Verb> <Adjective>
<Noun> → BITS | Students
<Verb> → is | are
<Adjective> → awesome | good



Variables or Non-Terminals

Constants or Terminals

Formal Grammar

- ✓A phrase structure grammar (or simply grammar) is (N,T, P, S) where
  - ✓N is a finite, non-empty set of Non-Terminals
  - ✓T is a finite, non-empty set of Terminals
  - ✓NnT =ϕ
  - ✓S is a special non-terminal (i.e S ∈N), called Start symbol
  - ✓P is a finite set whose elements are of the form, **α → β (N U T)**
    - ✓Elements – production rules

Formal Grammar

- ✓A phrase structure grammar (or simply grammar) is (N,T, P, S) where
  - ✓N is a finite, non-empty set of Non-Terminals
  - ✓T is a finite, non-empty set of Terminals
  - ✓NnT =ϕ
  - ✓S is a special non-terminal (i.e S ∈N), called Start symbol
  - ✓P is a finite set whose elements are of the form, **α → β (N U T)**
    - ✓Elements – production rules

Revision of CFG

- ✓Definition: A Context-Free Grammar consists of
  - ✓Non-terminals: Syntactic variables that help define the syntactic structure of the language.
  - ✓Terminals: The basic components found by the lexer. They are sometimes called token names, i.e., the first component of the token as produced by the lexer.
  - ✓Start Symbol: A nonterminal that forms the root of the parse tree.
  - ✓Productions:
    - ✓Head or left (hand) side or LHS. For context-free grammars, which are our only interest, the LHS must consist of just a single nonterminal.
    - ✓Body or right (hand) side or RHS. A string of terminals and non-terminals.

Notational Conventions

✓These symbols are terminals:

(a) Lowercase letters early in the alphabet, such as a, b, e.

(b) Operator symbols such as +, r, 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.

Notational Conventions

- ✓These symbols are nonterminals:

  (a) Uppercase letters early in the alphabet, such as A, B, C.

  (b) The letter S, which, when it appears, is usually the start symbol.

  (c) Lowercase, italic names such as *expr* or *stmt*.

  (d) When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs.

- ✓ For example, non- terminals for expressions, terms, and factors are often represented by E, T, and F, respectively.

Notational Conventions

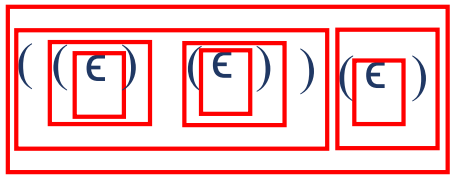
- ✓3. Uppercase letters late in the alphabet, such as X, Y, Z, represent grammar symbols; that is, either nonterminals or terminals.
- ✓4. Lowercase letters late in the alphabet, chiefly u, v, . . . , x, represent (possibly empty) strings of terminals.
- ✓5. Lowercase Greek letters, alpha, beta, gamma for example, represent (possibly empty) strings of grammar symbols
- ✓6. The head of the first production is the start symbol.

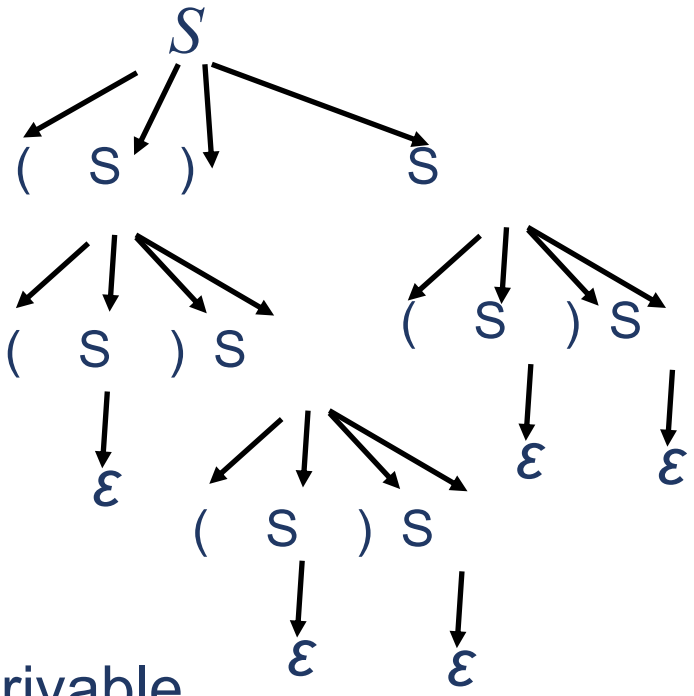
Generation of Derivation Tree

- ✓A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.
- ✓Representation Technique
- ✓**Root vertex** − Must be labeled by the start symbol.
- ✓**Vertex** − Labeled by a non-terminal symbol.
- ✓**Leaves** − Labeled by a terminal symbol or ε.

Grammar, Parse Tree, Derivations

- ✓Grammar (*G*) to match balanced parenthesis
- ✓ S → (S) S | ϵ
- ✓s = (()())() ∈ *L*(*G*)





- ✓*L*(*G*) = set of string derivable from S
- ✓Derivation (⇒)
  - ✓Process of generating a string derivable from S, by applying repeated productions.
  - ✓S ⇒ (S) S ⇒ ((S) S) S ⇒

Language of a Grammar

- ✓$L(G)$ = set of string derivable from S
- ✓Derivation ($\Rightarrow$) :
- ✓S $\Rightarrow$ (S) S $\Rightarrow$ ((S) S) S $\Rightarrow$ ((ϵ) S) S
  $\Rightarrow$ ((ϵ) (S) S) S $\Rightarrow$ (() (ϵ) S) S
  $\Rightarrow$ (() () ϵ) S $\Rightarrow$ (() ()) (S) S $\Rightarrow$ (() ()) (ϵ) S
  $\Rightarrow$ (() ()) () ϵ $\Rightarrow$ (() ()) () ϵ

- ✓$\Rightarrow$ Single Step Derivation
  - ✓((ϵ) (S) S) S $\Rightarrow$ (() (ϵ) S) S

- ✓$\overset{*}{\Rightarrow}$ Derivation in ≥ 0 steps
  - ✓((S) S) S $\overset{*}{\Rightarrow}$ (() (ϵ) S) S

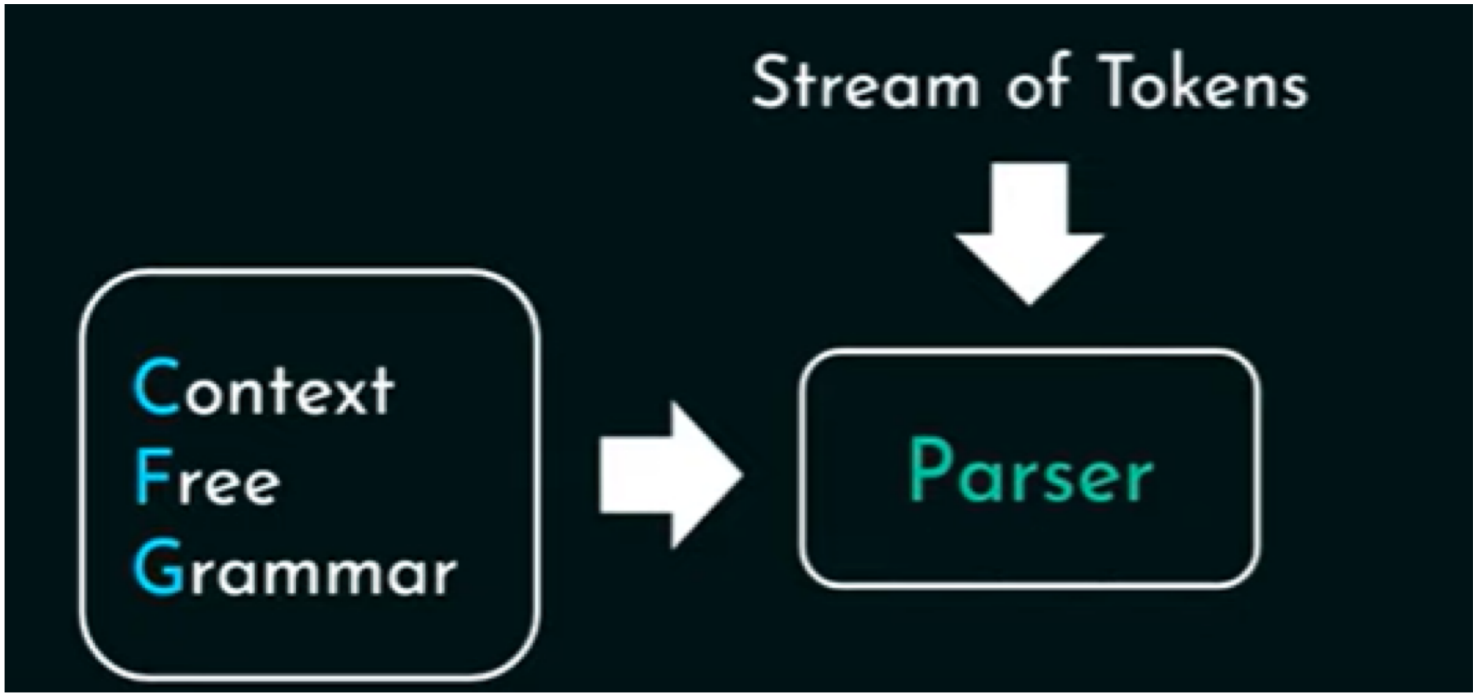
Recap

- ✓Syntax Analysis
  - ✓Introduction
  - ✓Context Free Grammars
  - ✓Derivation of CFG's

Overview

- ✓In this Lecture…
- ✓Syntax Analysis
  - ✓Introduction
  - ✓Context Free Grammars
    - ✓Leftmost & Rightmost Derivation of CFG's
    - ✓Derivation of Parse Tree

Syntax Analyzer



Formal Definition of a CFG

- ✓ A CFG consists of
  - ✓ A set of terminals T
  - ✓ A set of non-terminals N
  - ✓ A start symbol S (a non-terminal)
  - ✓ A set of productions

$$X \rightarrow Y_1 Y_2 \ldots Y_n$$

where $X \in N$ and $Y_i \in T \cup N \cup \{\varepsilon\}$

CFG :
LHS of any production/rule can only be a single non-terminal

Good enough for most programming lang constructs

- ✓ L(G) is the language of G
- ✓ if S is the start symbol, then L(G) is the set of all strings that can be generated (derivable) from S
  - ✓ $(a_1 \ldots a_n) | S \overset{*}{\Rightarrow} a_1 \ldots a_n$, all $a_i \in T$

Formal Definition of a CFG

E→ E+E | E * E | id

E→E + E
E→ E * E
E→ id

CFG→(N,T,P,S)

N={E}
T={+,*,id}

Generation of a String

- ✓Leftmost and Rightmost Derivation of a String
- ✓**Leftmost derivation** − A leftmost derivation is obtained by applying production to the leftmost variable in each step.
- ✓**Rightmost derivation** − A rightmost derivation is obtained by applying production to the rightmost variable in each step.

- ✓ Why need LM/RM derivations??
  - ✓ Leads to different parser implementations

Formal Definition of a CFG

E→ E+E | E * E | id

E→E + E
E→ E * E
E→ id

CFG→(N,T,P,S)

N={E}
T={+,*,id}

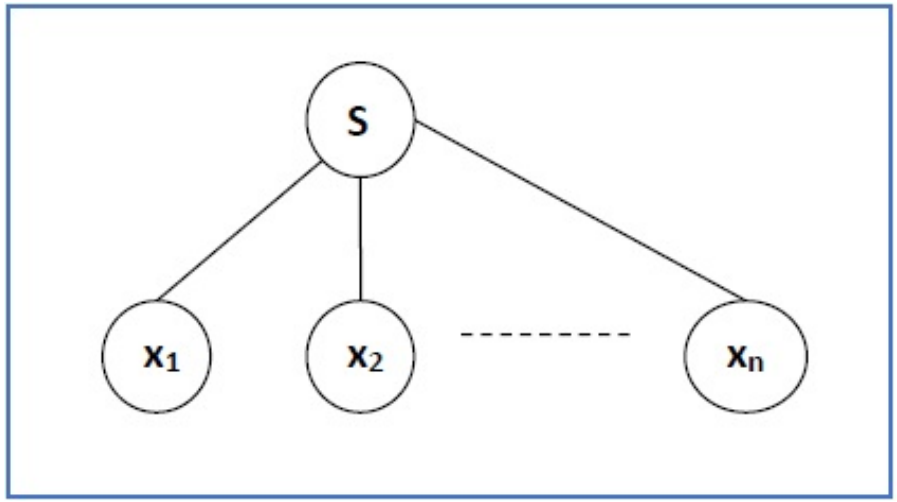
Example String: id + id * id

Leftmost Derivation:
E→E+E
E→id+E
E→id+E*E
E→id+id*E
E→id+id*id

Rightmost Derivation:
E→E+E
E→E+E*E
E→E+E*id
E→E+id*id
E→id+id*id

Generation of Derivation Tree

✓If S → x_1x_2 …… x_n is a production rule in a CFG, then the parse tree / derivation tree will be as follows −



Generation of Derivation Tree

- ✓There are two different approaches to draw a derivation tree

- ✓**Top-down Approach**
  - ✓Starts with the starting symbol **S**
  - ✓Goes down to tree leaves using productions
- ✓**Bottom-up Approach**
  - ✓Starts from tree leaves
  - ✓Proceeds upward to the root which is the starting symbol **S**

Formal Definition of a CFG

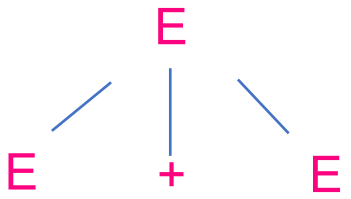
Example String: id + id * id

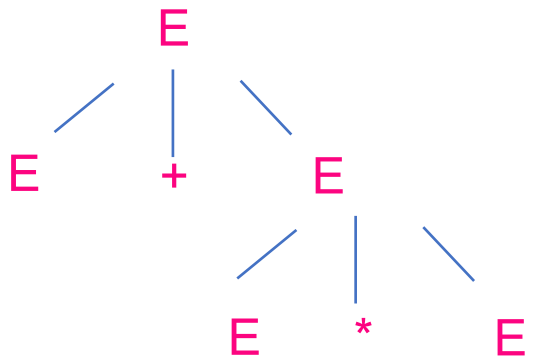
Leftmost Derivation:

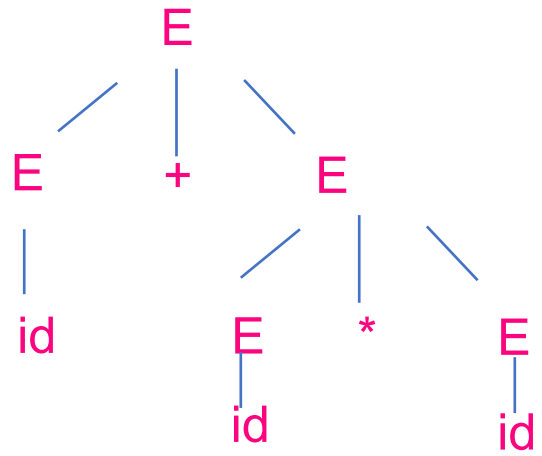
E→E+E
E→id+E
E→id+E*E
E→id+id*E
E→id+id*id

Rightmost Derivation:

E→E+E
E→E+E*E
E→E+E*id
E→E+id*id
E→id+id*id







Example

G: E → E+E | E*E | (E) | id | num

Input: (a + 23) * 12

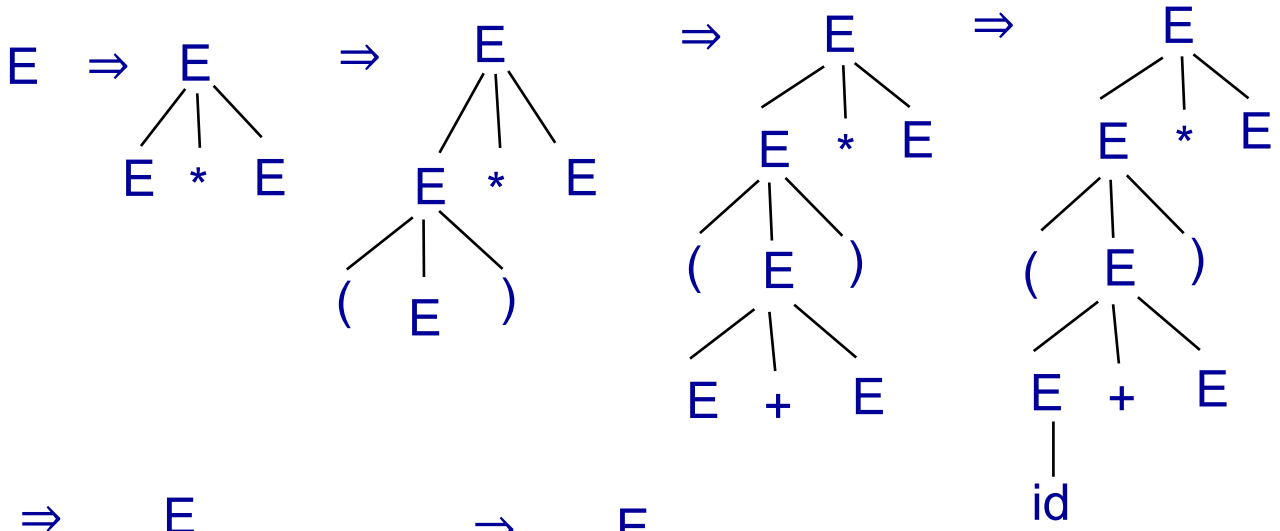
Leftmost Derivation

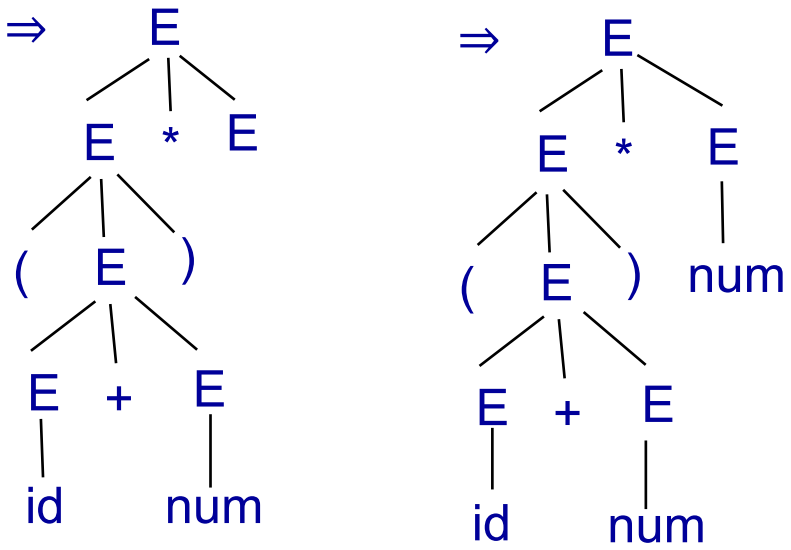
G: E → E+E | E* E | (E) | id | num

Input: (a + 23) * 12

Tokens: LP ID PLUS NUM RP MUL NUM

E ⇒ E * E
⇒ (E) * E
⇒ (E+E) * E
⇒ (id+E) * E
⇒ (id+num) * E
⇒ (id+num) * num





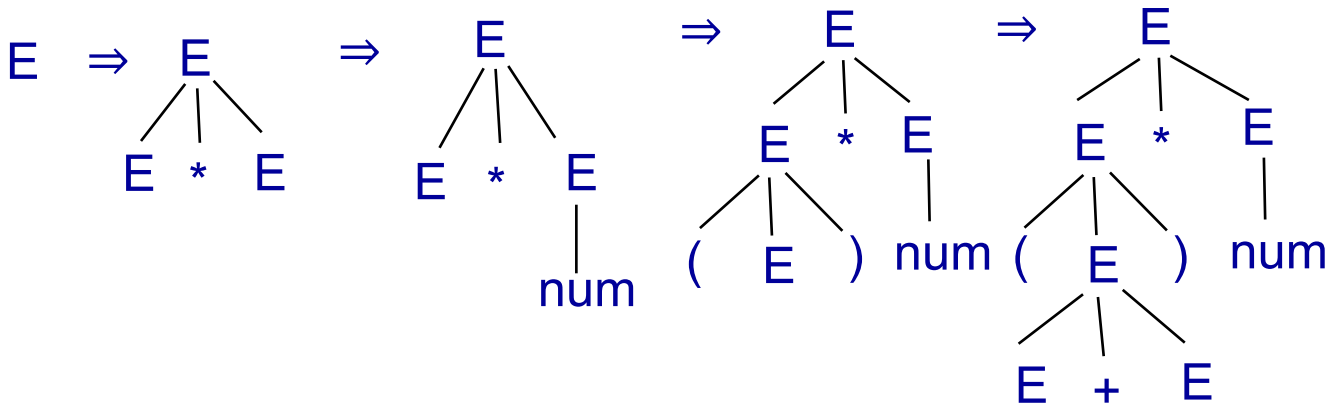
Rightmost Derivation

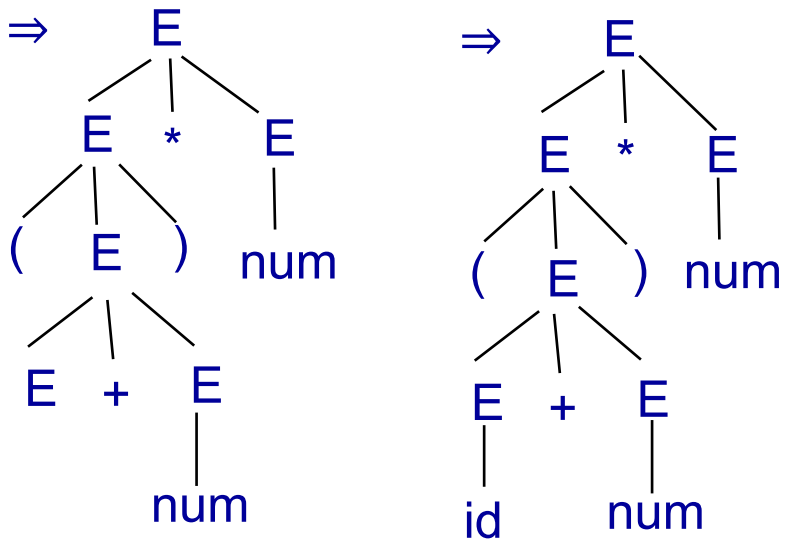
G: E → E+E | E* E | (E) | id | num

Input: (a + 23) * 12

Tokens: LP ID PLUS NUM RP MUL NUM

E ⇒ E * E
⇒ E * num
⇒ (E) * num
⇒ (E+E) * num
⇒ (E+num) * num
⇒ (id+num) * num





Why Derivations?

- ✓Syntax Analysis
  - ✓Check if the string s ∈ $L(G)$ (not sufficient for code gen)
  - ✓Also we need to generate parse tree (needed for code-gen)

- ✓ A derivation defines a parse tree
- ✓ But one parse tree may have many derivations.

- ✓ Leftmost and Rightmost derivations results in different parser implementations.
  - ✓ Leftmost -> Top-Down Parsing method
  - ✓ Rightmost -> Bottom-up Parsing method.

Example - Generation of Derivation Tree

- ✓**Example**
- ✓Let any set of production rules in a CFG be

  $X \rightarrow X+X \mid X*X \mid |X| \mid a$

  over an alphabet {a}.

Example - Generation of Derivation Tree

- ✓**Example**
- ✓Let any set of production rules in a CFG be

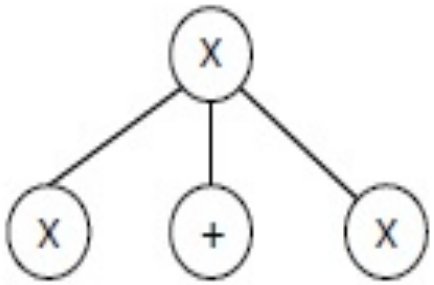
  $X \rightarrow X+X \mid X*X \mid X \mid a$

  over an alphabet {a}.
- ✓The leftmost derivation for the string **"a+a*a"** may be −

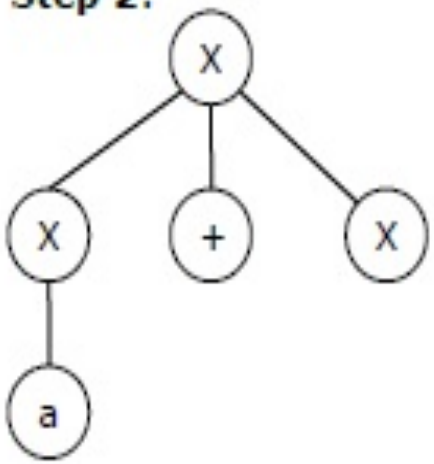
  $X \rightarrow X+X \rightarrow a+X \rightarrow a + X*X \rightarrow a+a*X \rightarrow a+a*a$
- ✓The stepwise derivation of the above string is shown as below −

Generation of Derivation Tree

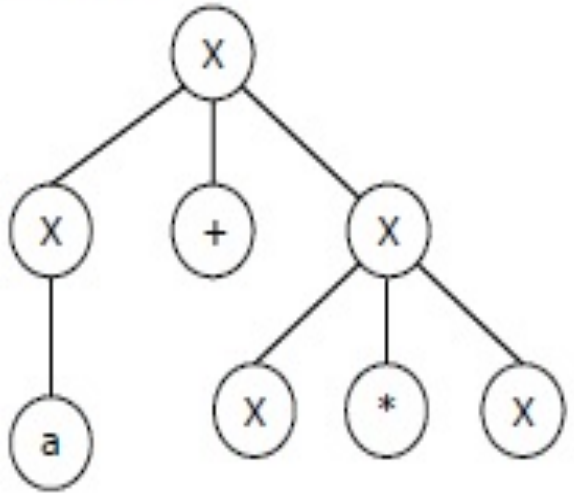
Step 1:



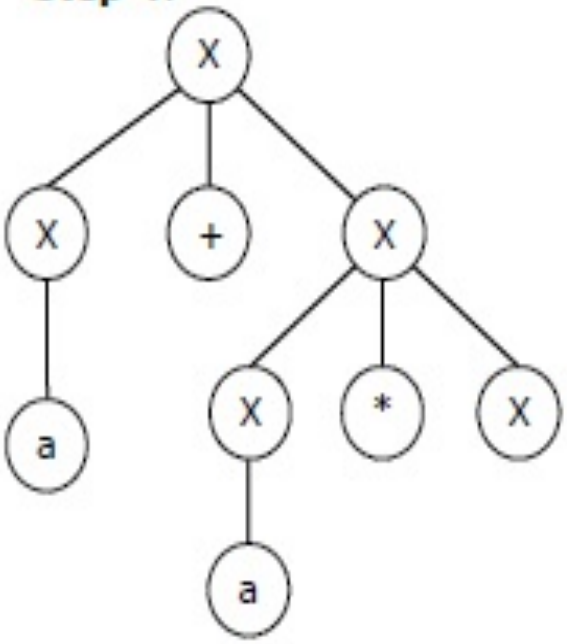
Step 2:



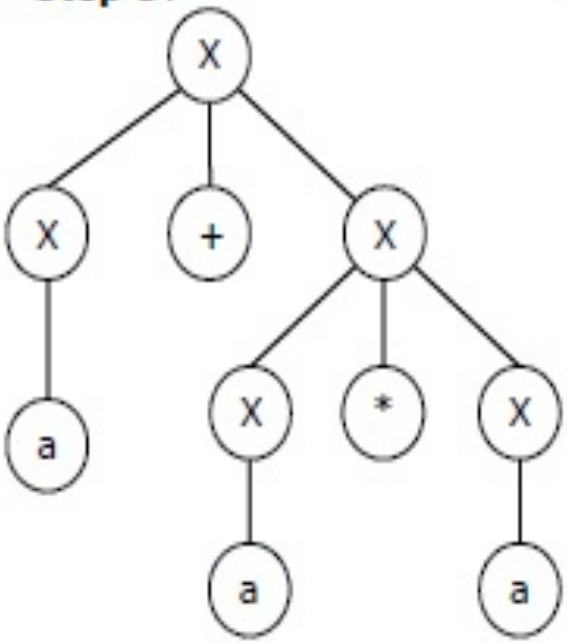
Step 3:



Step 4:



Step 5:

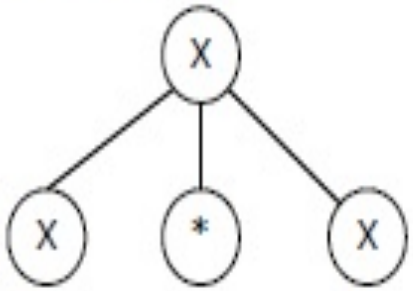


Generation of Derivation Tree

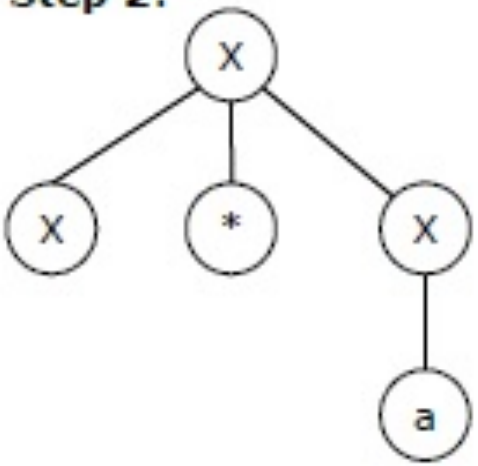
✓The rightmost derivation for the above string **"a+a*a"** may be

$$X \rightarrow X*X \rightarrow X*a \rightarrow X+X*a \rightarrow X+a*a \rightarrow a+a*a$$

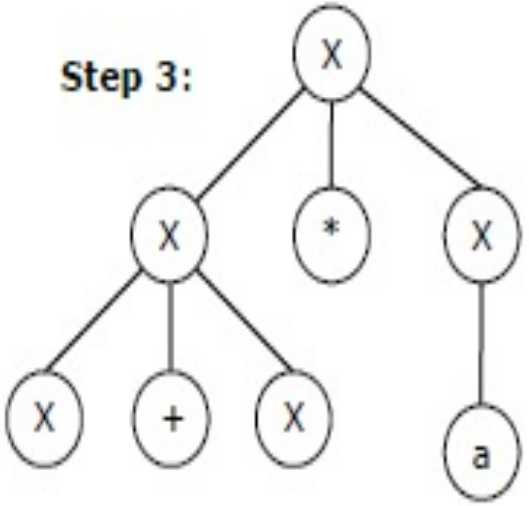
Step 1:



Step 2:



Step 3:

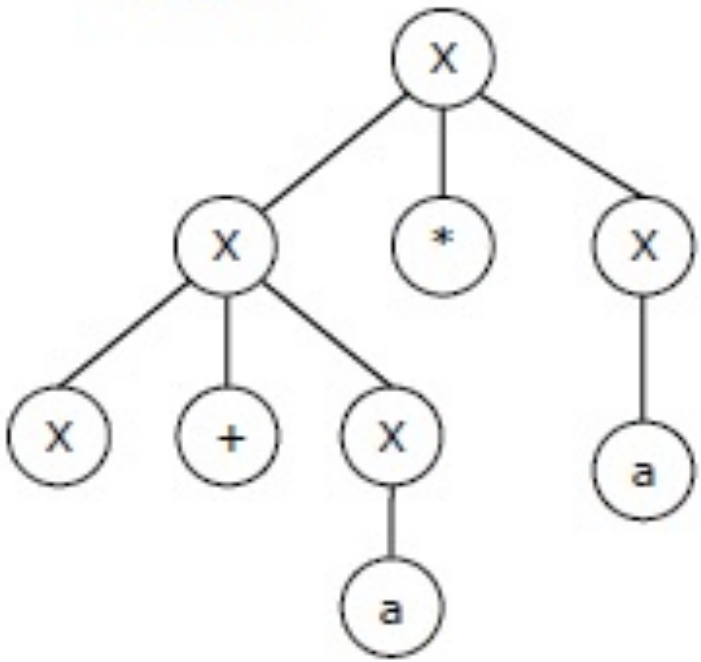


Generation of Derivation Tree

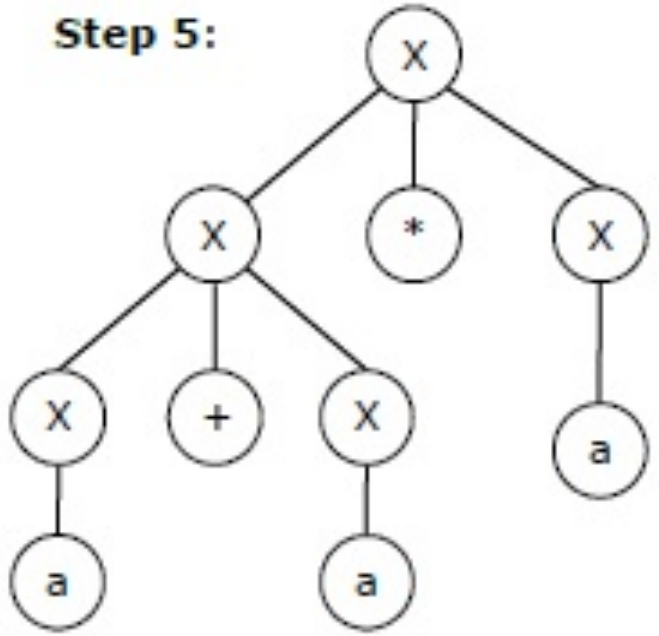
✓The rightmost derivation for the above string **"a+a*a"** may be

$$X \rightarrow X*X \rightarrow X*a \rightarrow X+X*a \rightarrow X+a*a \rightarrow a+a*a$$

Step 4:



Step 5:



Thank you

Recap

- ✓In last Lecture…
- ✓Syntax Analysis
  - ✓Introduction
  - ✓Context Free Grammars
    - ✓Leftmost & Rightmost Derivation of CFG's
    - ✓Derivation of Parse Tree

Overview

- ✓In this Lecture…
- ✓Syntax Analysis
  - ✓Introduction
  - ✓Context Free Grammars
    - ✓Leftmost & Rightmost Derivation of CFG's
    - ✓Derivation of Parse Tree
    - ✓LL and LR
  - ✓Using Ambiguous Grammars

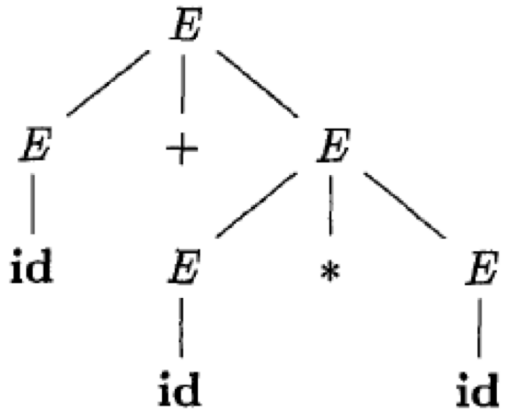
Ambiguity

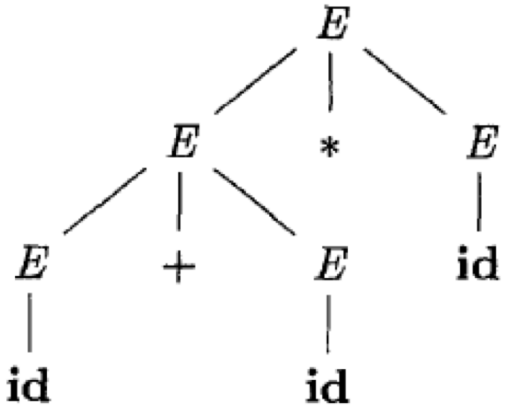
- ✓For some strings there exist more than one parse tree
  - ✓Or more than one leftmost derivation
  - ✓Or more than one rightmost derivation
- ✓Example: G: $E \rightarrow E+E \mid E*E \mid (E) \mid id \mid num$

  s: id+id*id

$$E \overset{lm}{\Rightarrow} E + E \overset{lm}{\Rightarrow} id + E \overset{lm}{\Rightarrow} id + E * E \overset{lm}{\Rightarrow} id + id * E \overset{lm}{\Rightarrow} id + id * id$$

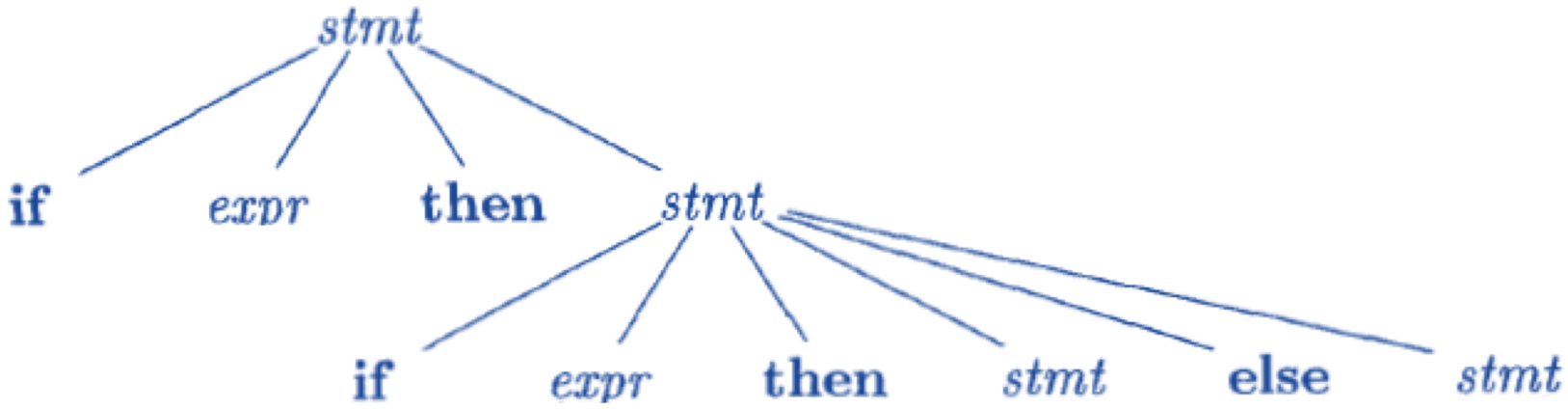
$$E \overset{lm}{\Rightarrow} E * E \overset{lm}{\Rightarrow} E + E * E \overset{lm}{\Rightarrow} id + E * E \overset{lm}{\Rightarrow} id + id * E \overset{lm}{\Rightarrow} id + id * id$$





Ambiguity (eg. dangling else problem)

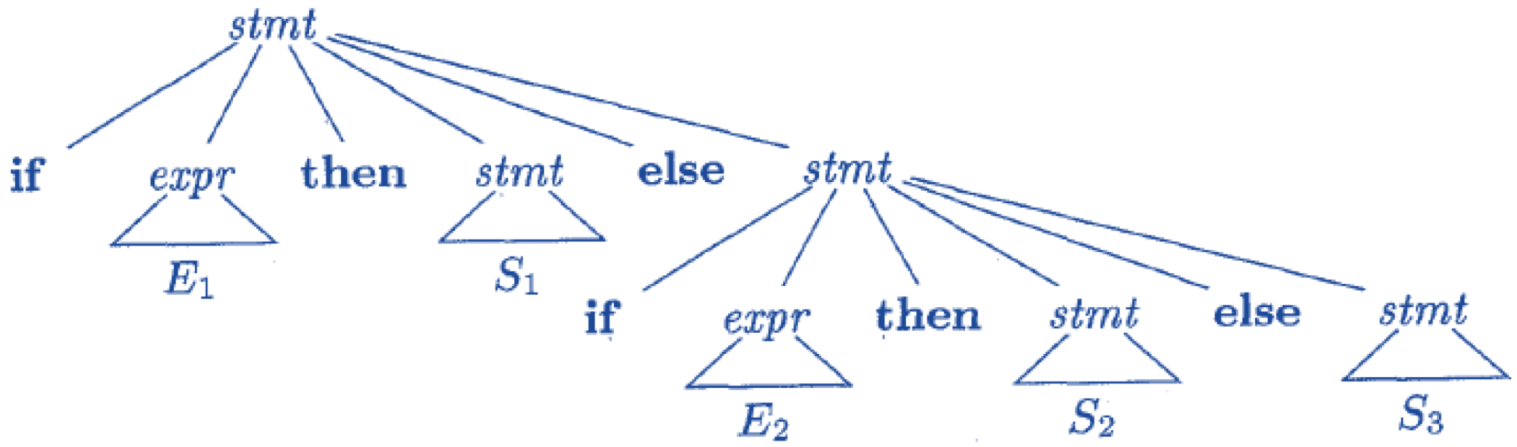
```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | other
```



Ambiguity (eg. dangling else problem)

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | other
```

✓**if E1 then S1 if E2 then S2 else S3**



Ambiguity (eg. dangling else problem)

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | other
```

✓This has two leftmost derivations for **`if E1 then if E2 then S2 else S3`**

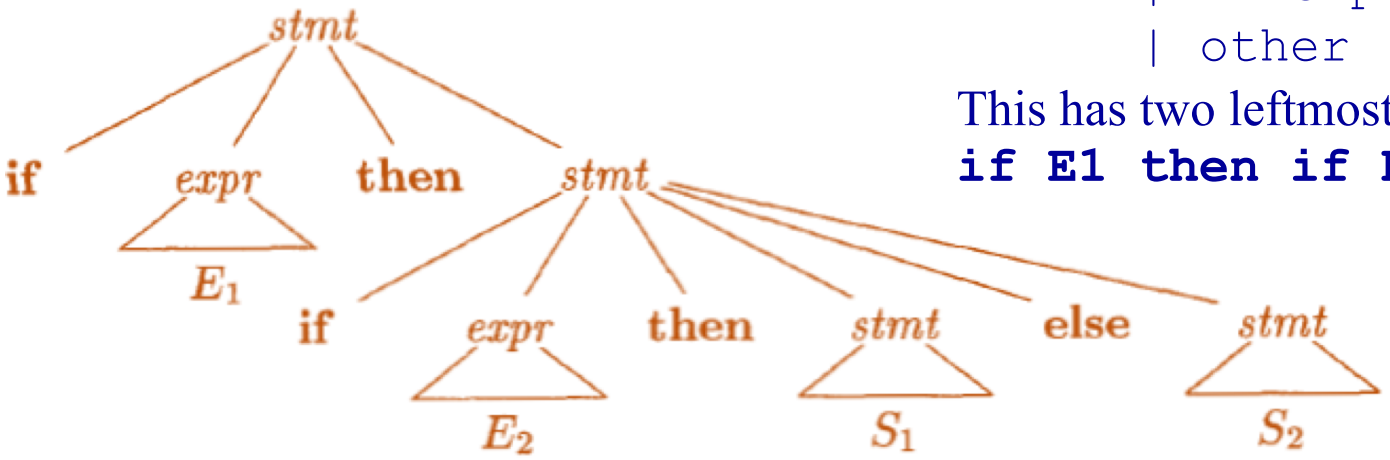
✓In this case we can find a non-ambiguous, equivalent grammar by rewriting the grammar.

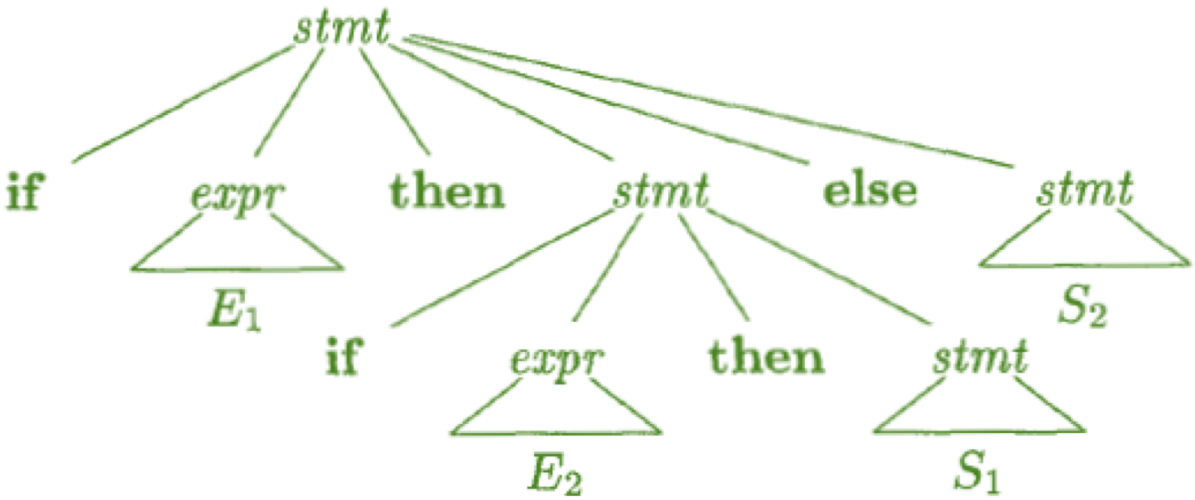
Ambiguity (eg. dangling else problem)

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | other
```

This has two leftmost derivations for

`if E1 then if E2 then S2 else S3`





Problems of Ambiguity

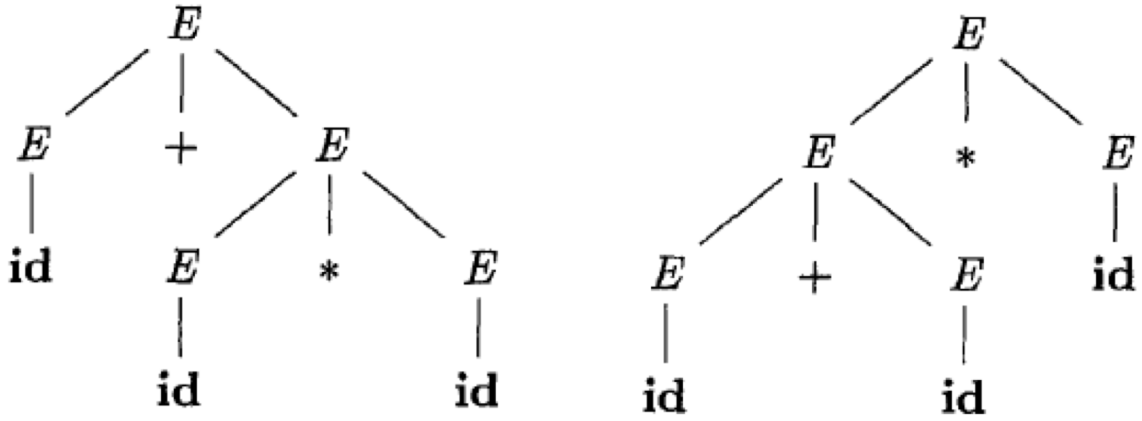
- ✓Operator Precedence Violation
- ✓Associativity Violation

Problems of Ambiguity

✓Operator Precedence Violation

E→E + E
E→ E * E
E→ id

String: id+id*id

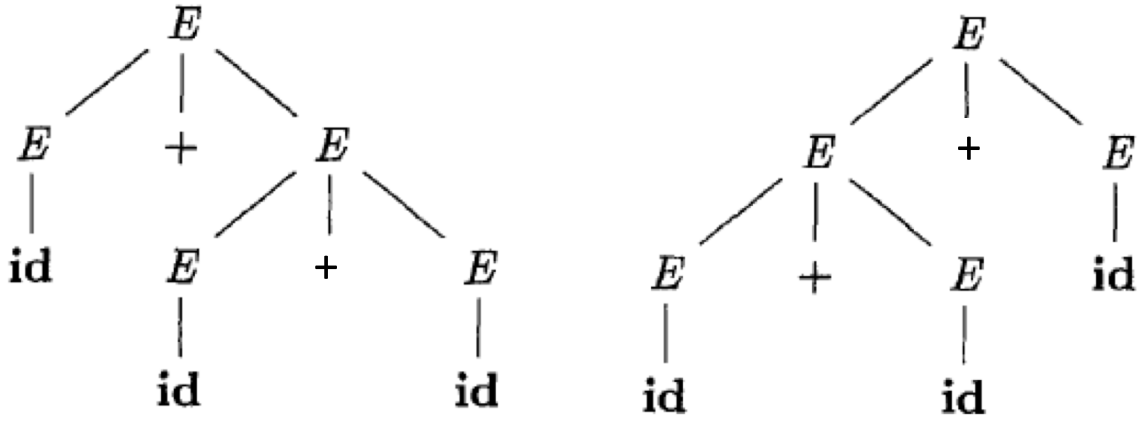


Problems of Ambiguity

✓Associativity Violation

E→E + E
E→ E * E
E→ id

String: id+id+id



Elimination of Ambiguity in Grammar

- ✓ No general techniques for handling ambiguity
- ✓ Can an ambiguous grammar be automagically be converted to an unambiguous one ?? Impossible
- ✓ Some techniques to disambiguation
  - ✓ Rewrite the grammar..(difficult..)
  - ✓ Use the natural (ambiguous) grammar.
    - ✓ Use tool provided disambiguation declaration
      - ✓ eg. yacc %left (to be seen later)

Elimination of ambiguity

- ✓Arithmetic Expression
  - ✓Rewrite the grammar.

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

  - ✓Use ambiguous grammar
    - ✓ $E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$
    - ✓Provide precedence declarations (provided by tool)
      - ✓%left + *
      - ✓%right =

Elimination of ambiguity

- ✓for `if` stmts
- ✓Idea:
  - ✓A statement appearing between a **then** and an **else** must be **matched**

```
stmt → if expr then stmt
     |if expr then stmt else stmt
     |other
```

This has two leftmost derivations for
`if E1 then if E2 then S2 else S3`

$$
\begin{array}{rcl}
stmt & \rightarrow & matched\_stmt \\
 & | & open\_stmt \\
matched\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } matched\_stmt \\
 & | & \textbf{other} \\
open\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } open\_stmt
\end{array}
$$

Recap

- ✓In this Lecture…
- ✓Syntax Analysis
  - ✓Introduction
  - ✓Context Free Grammars
    - ✓Leftmost & Rightmost Derivation of CFG's
    - ✓Derivation of Parse Tree
  - ✓Using Ambiguous Grammars
    - ✓Precedence & Associativity Violation
    - ✓Elimination of Ambiguity

Overview

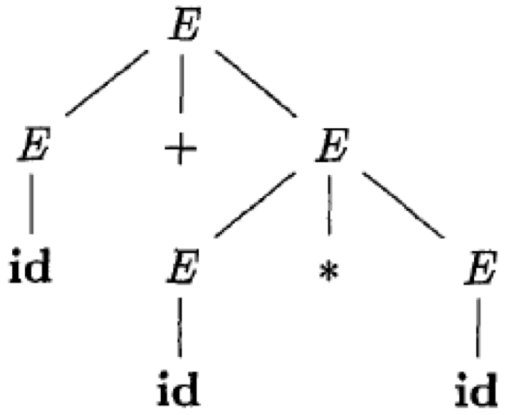
- ✓In this Lecture…
- ✓Syntax Analysis
  - ✓Introduction
  - ✓Context Free Grammars
    - ✓Leftmost & Rightmost Derivation of CFG's
    - ✓Derivation of Parse Tree
  - ✓Using Ambiguous Grammars
    - ✓Precedence & Associativity Violation
    - ✓Elimination of Ambiguity
    - ✓Elimination of Ambiguity – Precedence & Association
  - ✓Left & Right Recursive Grammar
    - ✓Problems of LR Grammar
    - ✓Solution

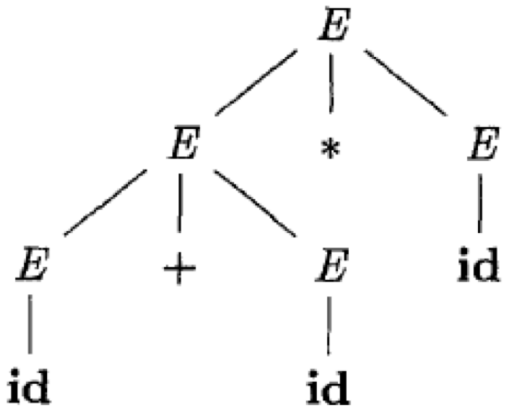
Elimination of ambiguity - Precedence

✓ Operator Precedence Violation

E→E + E
E→ E * E
E→ id

String: id+id*id

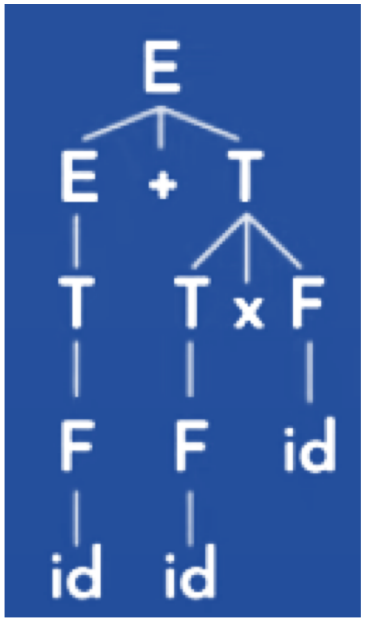




✓ Rewrite the Grammar

E→E + T | T
T→ T * F | F
F→ id

String: id+id*id

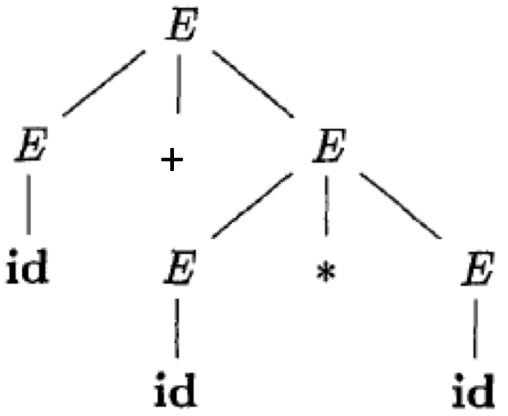


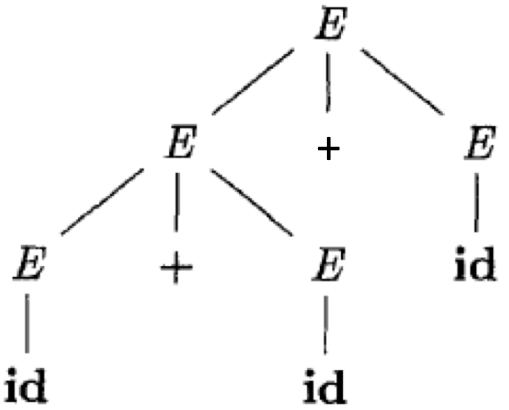
Elimination of ambiguity - Association

✓ Associativity Violation

E→E + E
E→ id

String: id+id+id

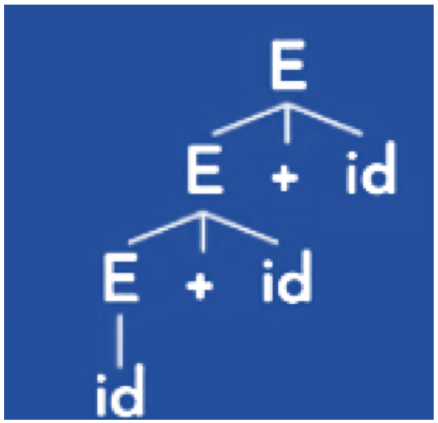




✓ Rewrite the Grammar as

E→E + id
E→ id

String: id+id+id (id+id)+id



✓ Left Associativity can be rectified with left recursive grammar

✓ Similarly right associativity can be rectified with right recursive grammar

Precedence & Association

✓ Grammar

E→E + T | T
T→ T * F | F
F→G^F | G
G→ id

✓ Example:

✓ Precedence - Left Associativity & right associativity

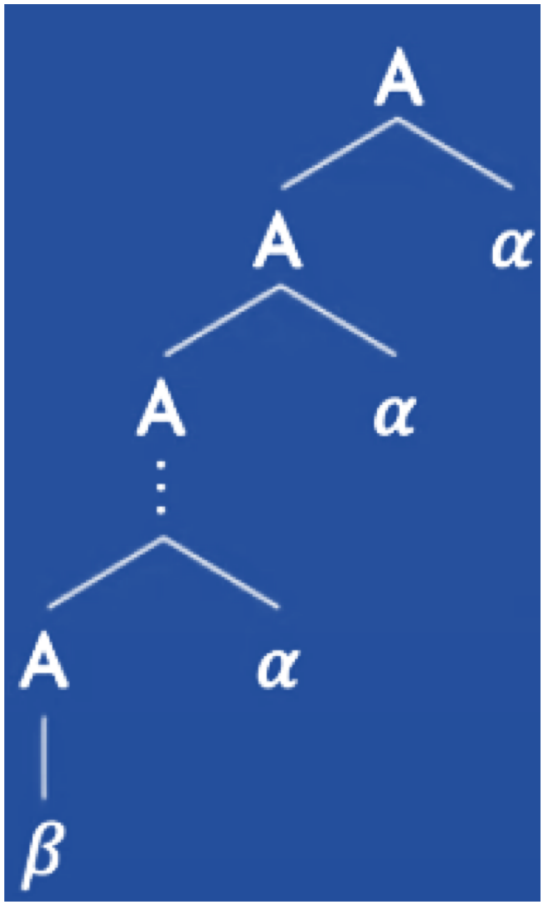
Left and Right Recursive Grammar

- ✓In a context-free grammar ***G***, if there is a production in the form ***A*** → ***A***α where ***A*** is a non-terminal and 'α' is a string of terminals, it is called a **left recursive** production. The grammar having a left recursive production is called a **left recursive grammar.**
- ✓In a context-free grammar ***G***, if there is a production is in the form ***A*** → α***A*** where ***A*** is a non-terminal and 'α' is a string of terminals, it is called a **right recursive** production. The grammar having a right recursive production is called a **right recursive grammar**.
- ✓Related to Parsing Techniques
  - ✓Top-down parser cannot work with Left recursive grammar
  - ✓Bottom-up parser prefers Left-recursive grammar.

Problems of Left Recursion

- ✓Left Recursive Grammar
- ✓A → ***A***α| β
- ✓Language A ⇒ βα*

```
A()
{
        A();
         α;
}
```



Problems of Left Recursion

- ✓Left Recursive Grammar
- ✓A → ***A***α| β
- ✓Language A ⇒ βα*

```
A()
{
        A();
         α;
}
```

- ✓Right Recursive Grammar
- ✓***A*** → αA| β
- ✓Language A ⇒ α*β

```
A()
{
        α;
        A();
}
```

Elimination of Left Recursion Problem

- ✓Left Recursive Grammar
- ✓$A \rightarrow A\ \alpha | \beta$
- ✓Language $A \Rightarrow \beta\alpha^*$

Solution:

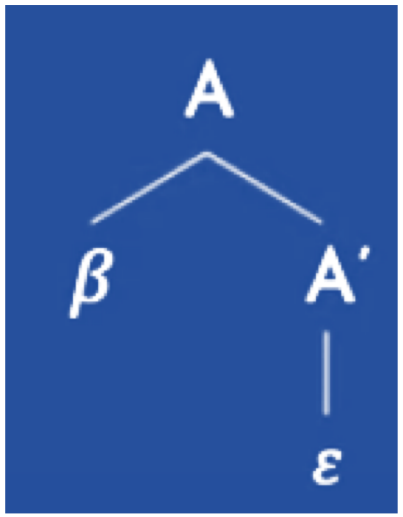
$A \rightarrow \beta\alpha^*$

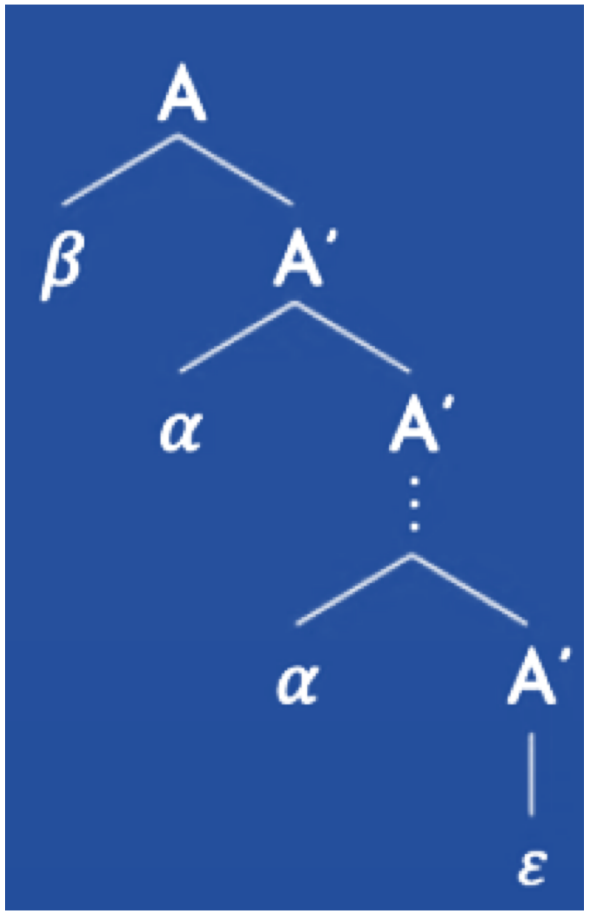
Rewrite as:

$A \rightarrow \beta A`$

$A` \rightarrow \alpha\ A` \mid \varepsilon$

- ✓Validate the rewritten grammar





Example 1

- ✓Eliminate the left recursion
- ✓P → P + Q | Q

Example 1

- ✓Eliminate the left recursion
- ✓P → P + Q | Q

- ✓Solution:
- ✓Consider rewriting this way

$$A \rightarrow \beta A`$$

$$A` \rightarrow \alpha A` \mid \varepsilon$$

- ✓Therefore

$$P \rightarrow QP`$$

$$P` \rightarrow +QP` \mid \varepsilon$$

Example 2

- ✓Eliminate the left recursion
- ✓S →S0S1S | 01

Example 2

- ✓Eliminate the left recursion
- ✓S →S0S1S | 01
- ✓Solution:
- ✓Consider rewriting this way

  $A \rightarrow \beta A`$

  $A` \rightarrow \alpha A` \mid \varepsilon$
- ✓Therefore

  S →01S`

  S`→0S1SS` | ε

Example 3

- ✓Eliminate the left recursion
- ✓A → (B)|b
- ✓B→B X A | A

Example 3

- ✓Eliminate the left recursion
- ✓A → (B)|b
- ✓B→B X A | A
- ✓Solution:
- ✓Consider rewriting this way

  $A \rightarrow \beta A`$

  $A` \rightarrow \alpha A` \mid \varepsilon$

- ✓Therefore

  A → (B)|b

  B→AB`

  B`→xAB`| ε

Thank you

Example 1

- ✓Determine whether the following grammar is suitable for predictive parsing

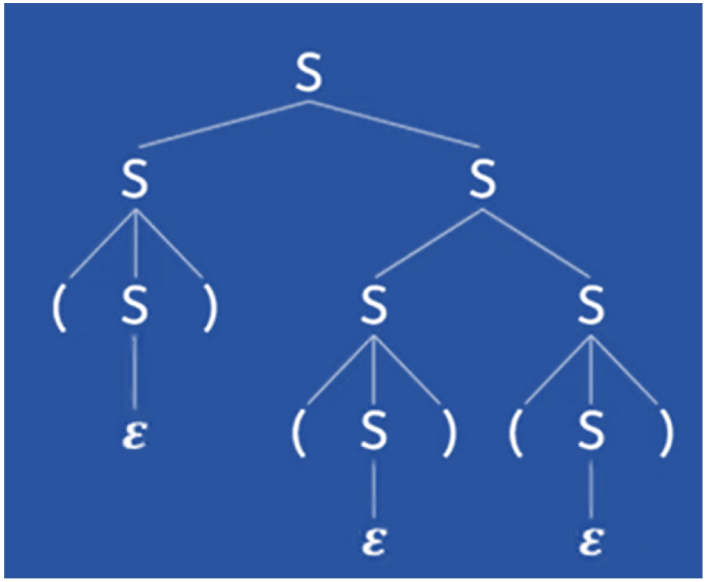
  $S \rightarrow (S)|SS|\varepsilon$

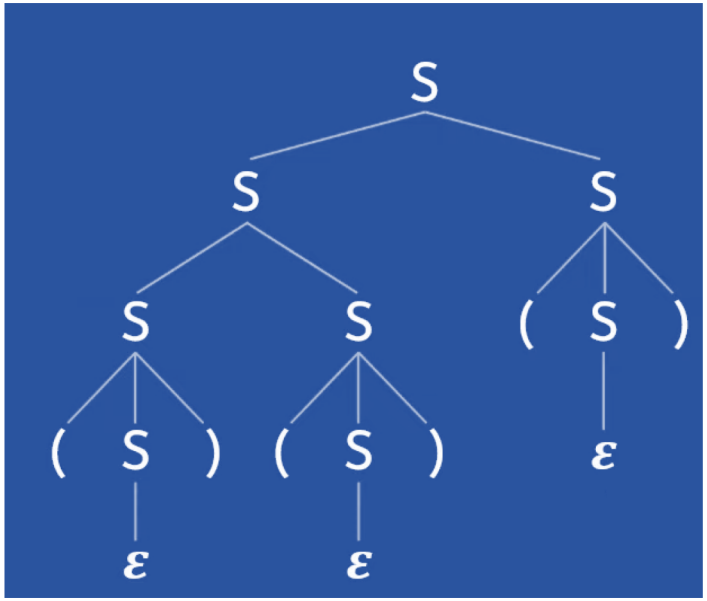
Example 1

✓Determine whether the following grammar is suitable for predictive parsing

S→(S)|SS|ε

Solution: ()()()





Parsing

- ✓ What does a parser need to decide?
  - ✓ Which production rule is to be used at each point of time ?
  - ✓ What is the next token?
    - ✓ Reserved word if, open parentheses, etc.
  - ✓ What is the structure to be built?
    - ✓ If statement, expression, etc.
  - ✓ The LL (top down) and LR (bottom-up) parsers are important in practice. Hand written parsers are often LL. Specifically, the predictive parsers are for LL grammars.
  - ✓ The LR grammars form a larger class. Parsers for this class are usually constructed with the aid of automatic tools.

Top–Down Parsing & Bottom–Up Parsing

- ✓ A parse tree is created from root to leaves
- ✓ The traversal of a parse tree is a pre-order traversal
- ✓ Tracing leftmost derivation
- ✓ Two types:
  - ✓ Backtracking parser
  - ✓ Predictive parser

- ✓ A parse tree is created from leaves to root
- ✓ The traversal of a parse tree is a reverse of post-order traversal
- ✓ Tracing rightmost derivation
- ✓ More powerful than top-down parsing

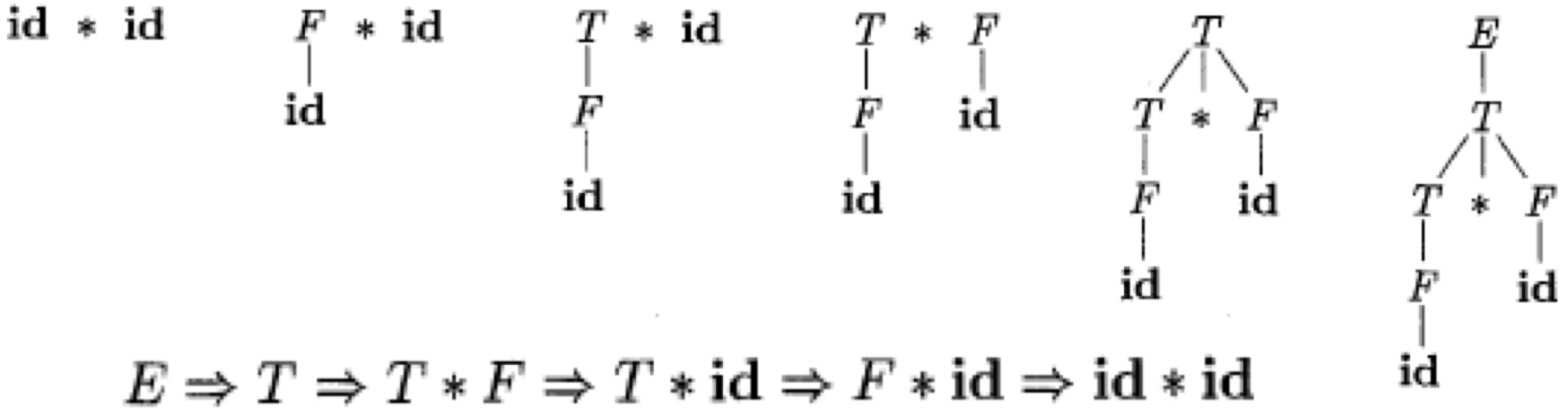
Bottom Up Parsing: Shift-Reduce parser

- ✓ The general idea is to shift some symbols of input to the stack until a reduction can be applied
- ✓ At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production
- ✓ The key decisions during bottom-up parsing are about when to reduce and about what production to apply
- ✓ A reduction is a reverse of a step in a derivation (RM)
- ✓ The goal of a bottom-up parser is to construct a derivation in reverse.

Bottom–Up Parsing: SR Parser

- ✓ Construction of the parse tree for an input string beginning at the leaves and working towards the root;
- ✓ Eg. bottom-up parse for id * id.

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (\, E \,) \mid \mathbf{id}
\end{aligned}
$$



$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

- ✓ This is a rightmost derivation applied in reverse.

Top–Down Parsing & Bottom–Up Parsing

- Derivation (Top-down)
  - Replace the nonterminal by its production rule
  - E => E + T etc.
  - Reach the input symbols

- Reduction (Bottom-up)
  - Reverse step of a derivation
  - Reduce id to F using production rule F->id
  - Apply a series of reduction to get to the start symbol.

Example

S → AB | aB

A → a | Aa

B → b

Input string "aab"

Handle & Handle Pruning

- ✓ Handle Pruning
  - ✓ Handle is substring which matches the body of production (i.e. RHS)
  - ✓ Reduce the handle by LHS of the production rule (1 step in the reverse of rightmost derivation)

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

Example

S → aABe

A → Abc|b

B → d

Input String: abbcde

Example

S → aABe

A → Abc|b

B → d

Input String: abbcde

Right Sentential Form	Handle	Reducing Production
abbcde	b	A→b
aAbcde	Abc	A→Abc
aAde	d	B→d
aABe	S	S→aABe
S		

Shift Reduce Parser

- ✓ Form of bottom-up parsing
  - ✓ Stack holds the grammar symbols
  - ✓ Input buffer holds the string to be parsed

STACK	INPUT
$	w $

  - ✓ Shift input symbols on to the stack
  - ✓ When a handle is found A->B, replace B by A on the stack
  - ✓ Repeat till all of input is scanned or error

STACK	INPUT
$ S	$

Shift Reduce Parser

- ✓ Actions:
  - ✓ Shift: Shift the next input symbol to the top of the stack
  - ✓ Reduce: The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string
  - ✓ Accept: Announce successful completion of parsing

STACK	INPUT
$ S	$

  - ✓ Error: Discover a syntax error and call an error recovery routine

Example

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (\, E \,) \mid \mathbf{id}
\end{aligned}
$$

- ✓ Shift Reduce Conflict
- ✓ Reduce Reduce Conflict

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \rightarrow \mathbf{id}$
$ $T * F$	$	reduce by $T \rightarrow T * F$
$ T	$	reduce by $E \rightarrow T$
$ E	$	accept

Example 2

Consider the following grammar

- ✓S→(L)|a
- ✓L→L,S|S

Parse the input string(a,(a,a)) using shift reduce parser

Example 2

Consider the following grammar

- ✓S→(L)|a
- ✓L→L,S|S

Parse the input string(a,(a,a)) using shift reduce parser

Stack	Input Buffer	Action
$	(a,(a,a))$	Shift
$(	a,(a,a))$	Shift
$(a	,(a,a))$	Reduce by S→a
$(S	,(a,a))$	Reduce by L→S
$(L	,(a,a))$	Shift
$(L,	(a,a))$	Shift
$(L,(	a,a))$	Shift
$(L,(a	,a))$	Reduce by S→a

Example 2

Consider the following grammar

- ✓S→(L)|a
- ✓L→L,S|S

Parse the input string(a,(a,a)) using shift reduce parser

Stack	Input Buffer	Action
$(L,(a	,a))$	Reduce by S→a
$(L,(S	,a))$	Reduce by L→S
$(L,(L	,a))$	Shift
$(L,(L,	a))$	Shift
$(L,(L,a	))$	Reduce by S→a
$(L,(L,S	))$	Reduce by L→L,S
$(L,(L	))$	Shift
$(L,(L)	)$	Reduce S→(L)

Example 2

Consider the following grammar

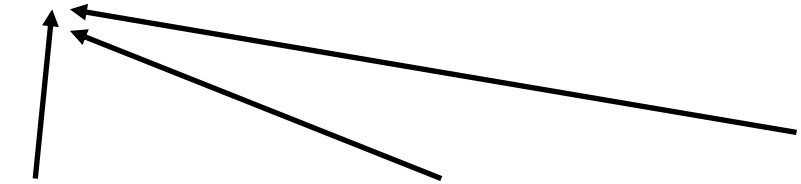
- ✓S→(L)|a
- ✓L→L,S|S

Parse the input string(a,(a,a)) using shift reduce parser

Stack	Input Buffer	Action
$(L,(L)	)$	Reduce S→(L)
$(L,S	)$	Reduce L→L,S
$(L	)$	Shift
$(L)	$	Reduce S→(L)
$S		

Conflicts During Shift-Reduce Parsing

- ✓ There are context-free grammars for which shift-reduce parsers cannot be used.
- ✓ Stack contents and the next input symbol may not decide action:
  - ✓ **shift/reduce conflict**: Whether make a shift operation or a reduction.
  - ✓ **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- ✓ If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.



left to right scanning | right-most derivation | k lookhead

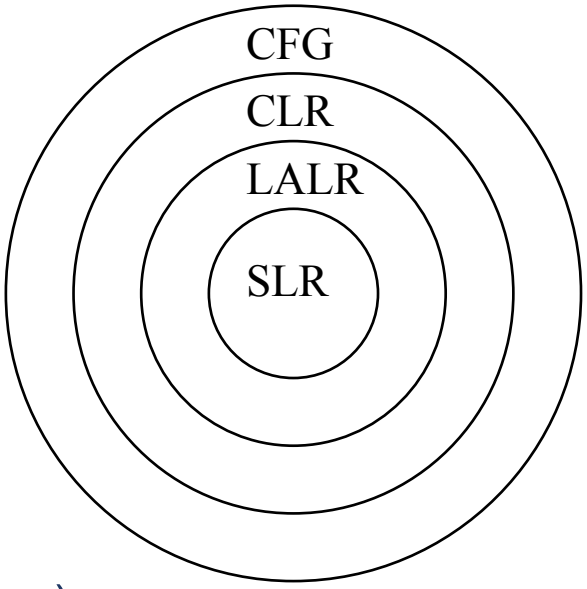
- ✓ An ambiguous grammar can never be a LR grammar.

Shift-Reduce Parsers

- ✓ There are two main categories of shift-reduce parsers

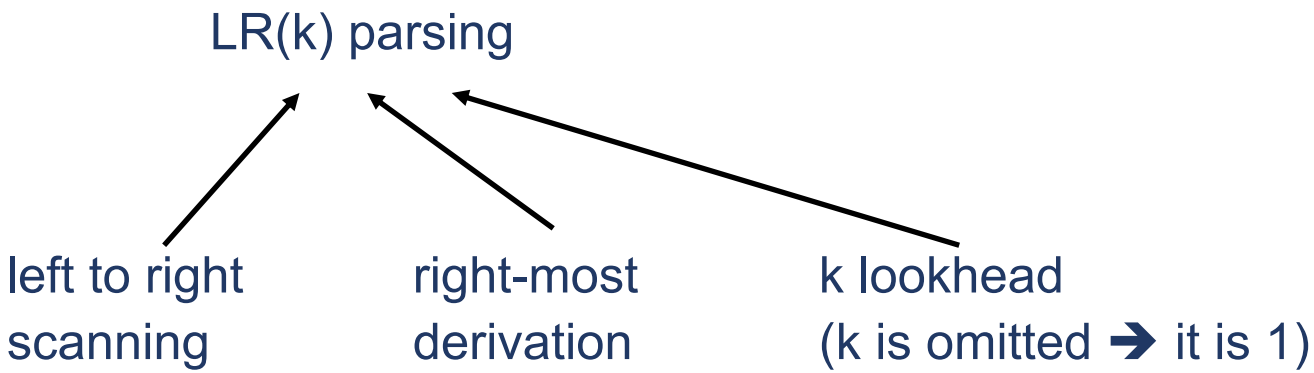
1. **Operator-Precedence Parser**
   - ✓ simple, but only a small class of grammars.

2. **LR-Parsers**
   - ✓ covers wide range of grammars.
     - ✓ **SLR – simple LR parser**
     - ✓ Canonical LR – most general LR parser
     - ✓ LALR – intermediate LR parser (lookhead LR parser)
   - ✓ SLR, CLR and LALR work same, only their parsing tables are different.



LR Parsers

- ✓ The most powerful shift-reduce parsing (yet efficient) is

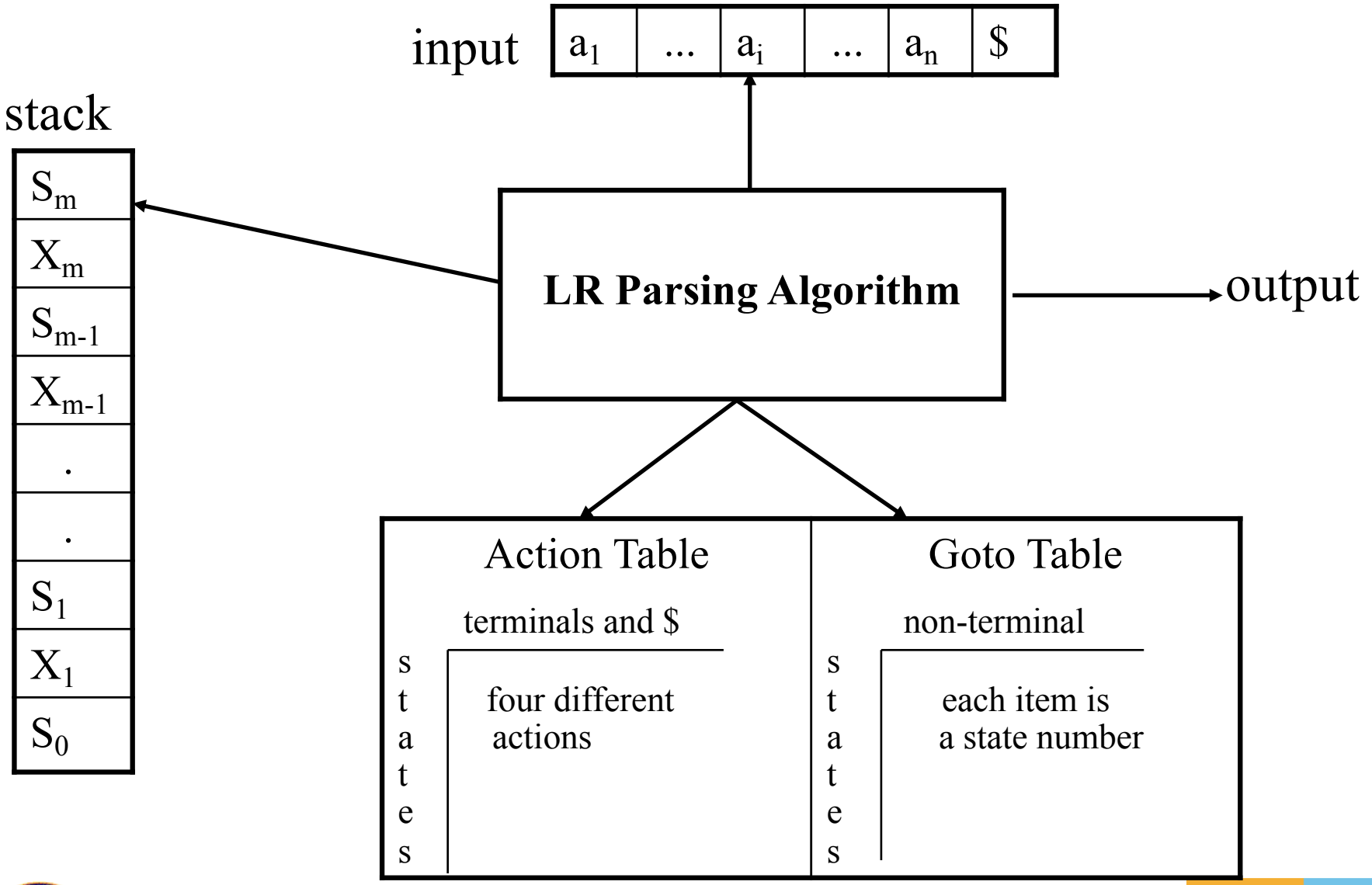


- ✓ LR parsing is attractive because:
  - ✓ LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - ✓ The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

$$\text{LL(1)-Grammars} \subset \text{LR(1)-Grammars}$$

  - ✓ An LR-parser can detect a syntactic error as soon as possible on a left-to-right scan of the input.

LR Parsing Algorithm



A Configuration of LR Parsing Algorithm

- ✓ A configuration of a LR parsing is:

$$(S_o X_1 S_1 \ldots X_m S_m, \; a_i a_{i+1} \ldots a_n \$)$$

Stack — Rest of Input

- ✓ S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)

- ✓ A configuration of a LR parsing represents the right sentential form:

$$X_1 \ldots X_m a_i a_{i+1} \ldots a_n \$$$

Actions of A LR-Parser

1. If action[S_m, a_i] = **shift s** -- shifts the current input symbol a_i and the next state **s**, which is given in action table entry action[S_m, a_i] onto the stack.

   (S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) ➔ (S_o X_1 S_1 ... X_m S_m a_i s, a_{i+1} ... a_n \$)

   a_{i+1} becomes the current input symbol.

2. If action[S_m, a_i] = **reduce A→β** (or **`rn`** where n is a production number)

   - ✓ pop 2|β| (=**r**, the length of β) items from the stack; **r** state symbols and **r** grammar symbols
   - ✓ then push **A** and **s** where **s=goto[s_{m-r},A],** which is given in the goto table entry.

   (S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) ➔ (S_o X_1 S_1 ... X_{m-r} S_{m-r} A s, a_i ... a_n \$)

   - ✓ Output is the reducing production reduce A→β
   - ✓ Current input symbol is not changed in a Reduce Move.

Actions of A LR-Parser

3. If action[S_m, a_i] = **Accept** – Parsing successfully completed

4. If action[S_m, a_i] = **Error** -- Parser detected an error (an empty entry in the action table). Parser calls an error recovery routine.

Example Grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Now we will see, actions of a (S)LR-Parser with an example input string:

id*id+id$.

We **assume** that (SLR) Parsing Table for Expression Grammar is **available**.

Later we will see how to construct the SLR Parsing Table.

(SLR) Parsing Tables for Expression Grammar

1) $E \rightarrow E+T$
2) $E \rightarrow T$
3) $T \rightarrow T*F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

	Action Table							Goto Table		
state	**id**	+	*	(	)	$		**E**	**T**	**F**
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

Actions of A (S)LR-Parser -- Example

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Constructing SLR Parsing Table
(of an augumented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'. $C \leftarrow \{I_0,\ldots,I_n\}$

2. Create the parsing **action table** as follows
   - If a is a terminal, $A \rightarrow \alpha.a\beta$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j***.
   - If $A \rightarrow \alpha.$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ for all **a** in **FOLLOW(A)** where $A \neq S'$.
   - If $S' \rightarrow S.$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR

3. Create the parsing **goto** table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser is the one constructed from closure[$S' \rightarrow .S$]

Constructing SLR Parsing Tables – LR(0) Item

- ✓ An **LR(0) item** of a grammar G is a production of G with a dot at the some position of the right side.
- ✓ Ex: $A \rightarrow aBb$ Possible LR(0) Items: $A \rightarrow .aBb$

  (four different possibility) $A \rightarrow a.Bb$

  $A \rightarrow aB.b$

  $A \rightarrow aBb.$
- ✓ Sets of **LR(0)** items will be the **states** of **action** and **goto** table of the SLR parser.
- ✓ A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- ✓ ***Augmented Grammar***:

  G' is G with a new production rule $S' \rightarrow S$ where S' is the new starting symbol.

Note: The production rule $A \rightarrow \varepsilon$ has just one possible item $A \rightarrow \cdot$

LR(0) ITEM

- ✓ An item indicates **how much** of a production we have seen at some point during parsing.
- ✓ For example, the item

  $E \rightarrow E \cdot + T$

- ✓ indicates that we have just seen a substring derivable from *E* and that next we hope to see one derivable from **+** *T*.
- ✓ An item is **reducible** if it has a dot at the rightmost position. For example:

$$T \rightarrow T * F \cdot$$

(**Intuition**: we have reached a point where we are ready to reduce

T * *F* to T)

Exception: an item defining the start symbol is **not** reducible.

Constructing an LR(0) automaton

✓ An LR(0) automaton is defined by two main functions:

- *GOTO(I, X)*: the LR(0) state reached by taking an *X* transition in state **I**.
- *STATES(G)*: the set of all LR(0) states of grammar *G*.

✓ To define these we need an auxiliary function called *CLOSURE.*

The Closure Operation

- ✓ If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from I by the two rules:

1. Initially, every LR(0) item in I is added to closure(I).
2. If $A \rightarrow \alpha.B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma$ will be in the closure(I).

We will apply this rule until no more new LR(0) items can be added to closure(I).

The Closure Operation -- Example

E' → E

E → E+T

E → T

T → T*F

T → F

F → (E)

F → id

closure({E' → ▪ E}) =

{ E' → ▪ E kernel item

E → ▪ E+T

E → ▪ T

T → ▪ T*F

T → ▪ F

F → ▪ (E)

F → ▪ id }

Kernel items: initial item and all items whose dots are not at the left end.

Non-Kernel item: these items have their dots at the left end.

Goto Operation in LR(0) Automaton

- ✓ If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then **goto(I,X)** is defined as follows:
  - ✓ If $A \rightarrow \alpha.X\beta$ in I then every item in **closure({$A \rightarrow \alpha X.\beta$})** will be in goto(I,X).

Example:

I ={ $E' \rightarrow .E$, $E \rightarrow .E+T$, $E \rightarrow .T$,
$T \rightarrow .T*F$, $T \rightarrow .F$,
$F \rightarrow .(E)$, $F \rightarrow .id$ }

goto(I,E) = { $E' \rightarrow E.$,
$E \rightarrow E.+T$
}

An item is **reducible** if it has a dot at the rightmost position.

Shortly, we will see how we can compute **goto(I,X)** for all the sets of LR(0) items.

Construction of The Canonical LR(0) Collection

- ✓ To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.

- ✓ ***Algorithm***:

  C is { closure({S'→.S}) }

  repeat the followings until no more set of LR(0) items can be added to ***C***.

    for each I in ***C*** and each grammar symbol X

      if goto(I,X) is not empty and not in ***C***

        add goto(I,X) to ***C***

- ✓ **goto** function is a DFA on the sets in C.

Example

Grammar:
E' → E
E → E+T | T
T → T*F | F
F → (E) | id

I_0: closure(E'→.E)
E' → .E
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id

I_1: goto(I_0,E) E' → E.
E → E. + T

I_2: goto(I_0,T)
E → T.
T → T. *F

I_3: goto(I_0,F)
T → F.

I_4: goto(I_0,( )
F → (.E)
E → .E + T E → .T
T → .T * F
T → .F
F → .(E)
F → .id

I_5: goto(I_0,id)
F → id.

Example

I_6: goto(I_1,+)
E → E + .T
T → .T * F
T → .F
F → .(E)
F → .id

I_7: goto(I_2,*)
T → T * .F
F →.(E)
F → .id

I_8: goto(I_4,E)
F → (E.)
E → E. + T

goto(I_4,T) is I_2
goto(I_4,F) is I_3
goto(I_4,( ) is I_4
goto(I_4,id) is I_5

I_9: goto(I_6,T)
E → E + T.
T → T. * F

goto(I_6,F) is I_3
goto(I_6,( ) is I_4
goto(I_6,id) is I_5

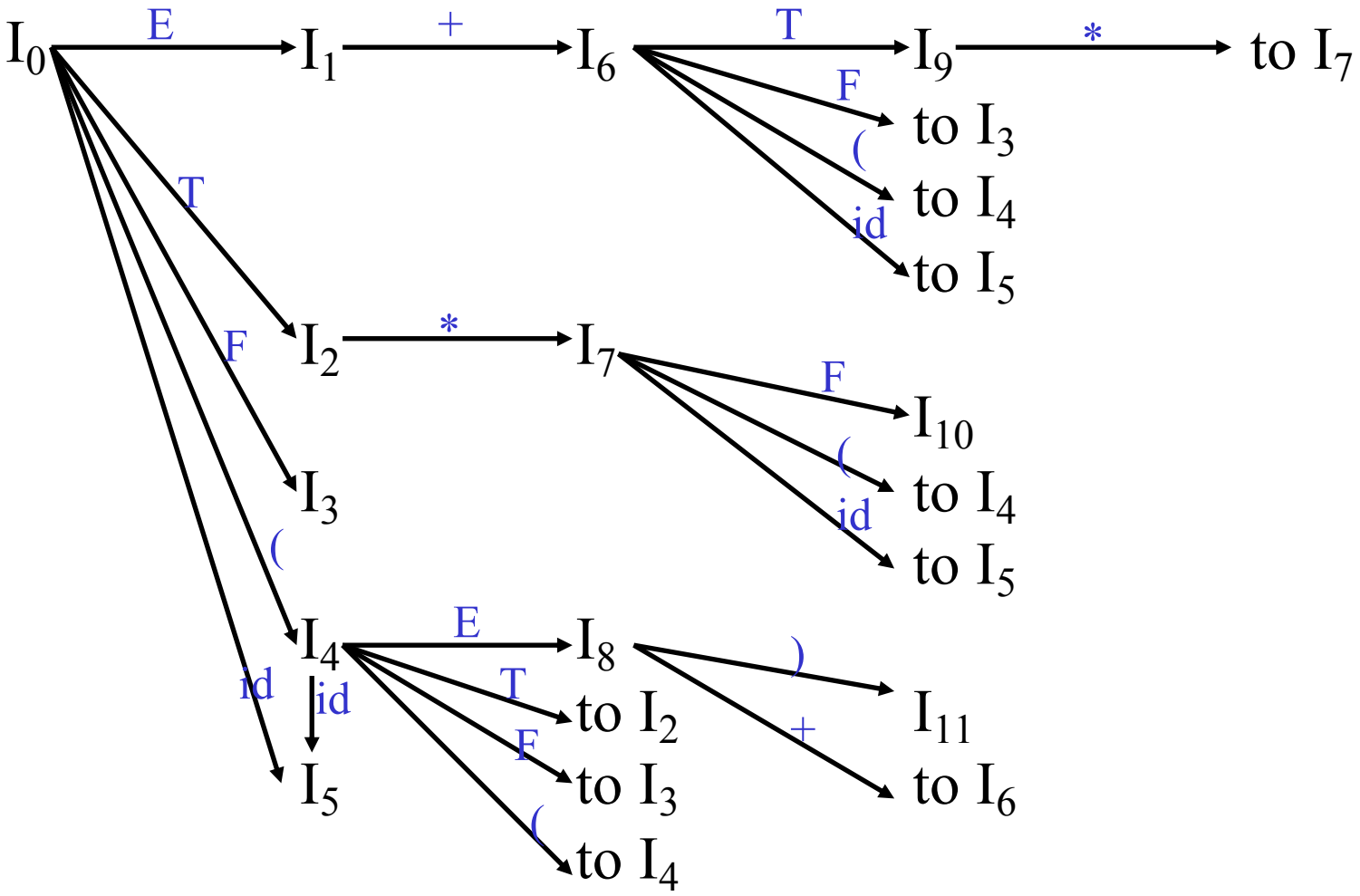
I_{10}: goto(I_7,F)
T → T * F.

goto(I_7,( ) is I_4
goto(I_7,id) is I_5

I_{11}: goto(I_8,))
F → (E).

goto(I_8,+) is I_6
goto(I_9,*) is I_7

Transition Diagram (DFA) of Goto Function



Compute FIRST for Any String X

- ✓If X is a terminal symbol
  - ➔ FIRST(X)={X}
- ✓If X is a non-terminal symbol and $X \rightarrow \varepsilon$ is a production rule
  - ➔ ε is in FIRST(X).
- ✓If X is a non-terminal symbol and $X \rightarrow Y_1Y_2..Y_n$ is a production rule
  - ➔ if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then **a** is in FIRST(X).
  - ➔if ε is in all FIRST(Y_j) for j=1,...,n for j=1,...,n then ε is in FIRST(X).

FIRST

E’ → E
E → E+T | T
T → T*F | F
F → (E) | id

FIRST(E′) = {(, id}
FIRST(E) = {(, id}
FIRST(T) = {(, id}
FIRST(F) = {(, id}

Compute FOLLOW (for non-terminals)

- ✓If S is the start symbol
  - ➔ $ is in FOLLOW(S)
- ✓if $A \rightarrow \alpha B \beta$ is a production rule
  - ➔ everything in FIRST(β) is FOLLOW(B) except ε

- ✓If ($A \rightarrow \alpha B$ is a production rule) or
  ($A \rightarrow \alpha B \beta$ is a production rule and ε is in FIRST(β))
  ➔ everything in FOLLOW(A) is in FOLLOW(B)

We apply these rules until nothing more can be added to any follow set.

FOLLOW

E’ → E
E → E+T | T
T → T*F | F
F → (E) | id

FOLLOW(E’) = {$}
FOLLOW(E) = {+,),$}
FOLLOW(T) = {+,),$,*}
FOLLOW(F) = {+,),$,*}

SLR Parse Table of Expression Grammar

	Action Table							Goto Table		
state	**id**	+	*	(	)	$		**E**	**T**	**F**
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

SLR(1) Parsing Table

- ✓ The parsing table consisting of the ACTION and GOTO functions determined by the (previous) algorithm is called the SLR(1) table for G
- ✓ An LR parser using the SLR(1) table for G is called the SLR(1) parser for G
- ✓ A grammar having an SLR(1) parsing table is said to be SLR(1)
- ✓ Usually the 1 after the SLR is omitted since we shall not deal here with parsers having more than one symbol of lookahead
- ✓ Every SLR(1) grammar is unambiguous

Example: Unambiguous grammar that is not SLR(1)

- ✓Consider the following Grammar G. Compute the Canonical LR(0) collection for the grammar G. Also construct the parse table for the grammar. Verify if the grammar is in SLR(1)

$$
\begin{array}{lcl}
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
$$

Example: Unambiguous grammar that is not SLR(1)

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S \cdot$

I_2: $S \to L \cdot = R$
$R \to L \cdot$

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id} \cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_7: $L \to * R \cdot$

I_8: $R \to L \cdot$

I_9: $S \to L = R \cdot$

Figure 4.39: Canonical LR(0) collection for grammar (4.49)

Consider the set of items I2. The first item in this set makes ACTION[2, =] be shift 6

Since FOLLOW(R) contains =, the second item sets ACTION[2, =] to reduce R → L

Since there is both a shift and a reduce entry in ACTION[2; =], state 2 has a shift/reduce conflict on input symbol =

Thank you

innovate achieve lead

BITS Pilani
Dubai Campus

Time for Tutorial

Tutorial III

Question 1

- ✓Consider the Grammar

  $E \rightarrow E + E$

  $E \rightarrow E * E$

  $E \rightarrow (E)$

  $E \rightarrow id$

- ✓Perform Rightmost Derivation string $id_1 + id_2 * id_3$. Find Handles at each step.

Question 2

- ✓Consider the following Grammar

  S → CC

  C → cC

  C → d

  - ✓Check whether input string "ccdd" is accepted or not accepted using Shift-Reduce parsing.

- ✓Consider the following grammar and eliminate left recursion-

  A → ABd / Aa / a

  B → Be / b