Syntax Analysis

Shyamalendu Kandar
Assistant Professor,
Department of Information Technology
IIEST, Shibpur

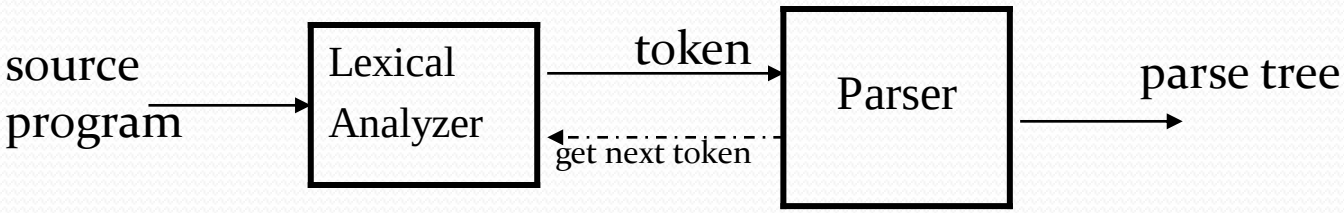
Syntax Analyzer

- *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)*.
- 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 Role of Parser

- Parser works on a stream of tokens, generated by lexical analyzer.

- The smallest item is a token.
- It verifies that the string can be generated by the grammar for the source language



Parsers (cont.)

- We categorize the parsers into two groups:

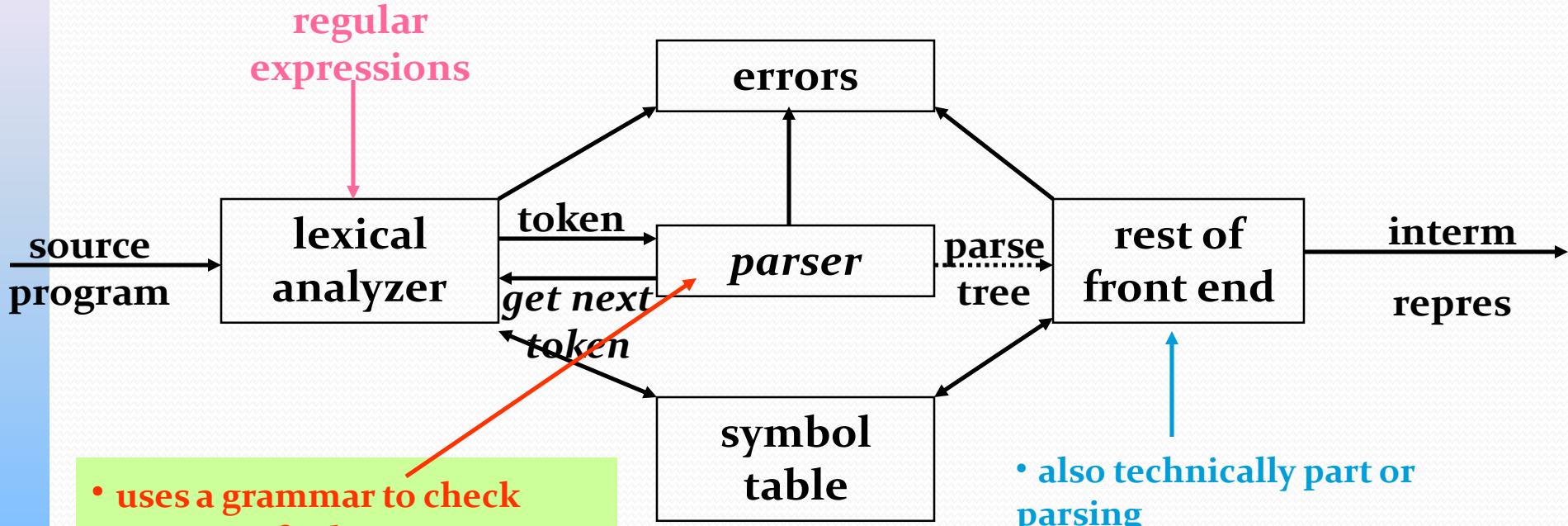
1. **Top-Down Parser**
   - the parse tree is created top to bottom, starting from the root.
2. **Bottom-Up Parser**
   - the parse is created bottom to top; starting from the leaves

- 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

Role Of Parser

- To identify the language constructs present in a given input program. If the parser determines the input to be valid one, it outputs a representation of the input in the form of a parse tree.
- If the input is grammatically incorrect, the parser declares the detection of syntax error in the input. This case no parse tree can be produced.

Parsing During Compilation



- uses a grammar to check structure of tokens
- produces a parse tree
- syntactic errors and recovery
- recognize correct syntax
- report errors

- also technically part or parsing
- includes augmenting info on tokens in source, type checking, semantic analysis

Parsing During Compilation

For parsing a string by a Grammar, general parsing methods are

- CYK(Cocke-Younger-Kasami) Method.
- Earley's Method.

CYK Algorithm.

Question: whether a particular string is generated by the grammar or not..

- A polynomial time algorithm called CYK algorithm can be constructed to check this.
- Proposed by John Cocke, Daniel Younger, and Tadao Kasami.
- According to the algorithm, the string of terminals is generated from length 1 to l.
- l is the length of the string w, the string to be checked for membership.
- If w ∈ the set of string generated, then w is a member of the strings generated by the CFG.
- For this, we need to convert the given grammar into CNF.

CYK Algo.

function CKY (word w, grammar P) returns table

for i ← from 1 to LENGTH(w) do

  table[i-1, i] ← {A | A → w_i ∈ P }

for j ← from 2 to LENGTH(w) do

  for i ← from j-2 down to 0 do

    for k ← i + 1 to j – 1 do

      table[i,j] ← table[i,j] ∪ {A | A → BC ∈ P,

        B ∈ table[i,k], C ∈ table[k,j] }

If the start symbol *S* ∈ table[0,n] then w ∈ L(G)

Assignment

Find the time and space complexity of CYK algorithm.

Ans: Time $O(n^3)$

Space $O(n^2)$

Earley's Method

Proposed by Jay Earley.

top-down dynamic programming algorithm.

The Earley parser executes in cubic time in the general case .

In Compiler Design

- CYK and Earley's Methods are inefficient to use in Compiler Design.
- Methods used in Compiler Design are
  - ✓ Top Down Parsing
  - ✓ Bottom Up Parsing.

In this section we will learn LL (Top-Down) and LR (Bottom Up).

❑In this section we assume that the output of the parser is some representation of the parse tree from the tokens produced by Lexical Analyzer.

❑In practice there are number of tasks might be conducted during parsing such as

- ✓ Collecting information about various tokens into the symbol table.
- ✓ Performing Type checking
- ✓ Generating intermediate code

These are mentioned as *rest of front ends*

Error Handling

- Error handling is one of the most important feature of any modern compiler.
- It is highly unlikely that a compiler without good error handling capacity will be accepted by the users even if it can produce correct code for the correct program.
- The most important challenge , here is to have a good guess of possible mistakes that a programmer can do and to come up with strategies to point those errors to the user in a very clear and unambiguous manner.
- Most programming language specification do not describe how a compiler should response to an error.
- It is left to the compiler designer.

Error Handling Contd.

- The common errors occurring in the programs can be classified into four categories.
  - Lexical error: These errors are mainly spelling mistakes and accidental insertion of foreign characters, for e.g.. '$', if the language does not allow it. They are mostly caught by the lexical analyzer.
  - Syntactic error: These are grammatical mistakes , such as unbalanced parentheses in arithmetic expressions. The parser should be able to catch these errors efficiently.
  - Semantic error: They involves errors due to undefined variables, incompatible operands to an operator etc. These errors can be caught by introducing some extra check during parsing.
  - Logical error: These are errors such as infinite loops. There is not any way to catch the logical error automatically. However use of debugging tools may help the programmer to identify such errors.

Error Recovery Strategies

- Thus an important challenge of syntax analysis phase is to detect syntactical errors.
- However nobody would like a compiler that stops after detecting the first error because, there may be many more such errors.
- If all or most of these errors can be reported to the user in a single go, the user may correct all of them and resubmit for compilation.
- But in most of cases, the presence of error in input stream leads the parser into an erroneous state, from where it can't proceed further until certain portion of its work is undone.
- The strategies involved in the process broadly known as error recovery strategies.

Goal of error Handler

- It should report the presence or error cleanly & accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct program.

Error Recovery Strategies

Panic Mode– In this case the parser discards enough number of tokens to reach a descent state on the detection of an error.

On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronized token is found.

```
a=b + c      // no semi-colon
d=e + f ;
```

The synchronizing tokens are usually delimiters, such as semicolon or end.

A set of tokens marking the end of language constructs is define to make a synchronizing set { e.g. ';' , '}' }

Discard tokens until a "synchro" token is found (end, ";", "}", etc.)

-- Decision of designer

-- Problems:

skip input ⇒miss declaration ⇒ causing more errors ⇒miss errors in skipped material

-- Advantages:

simple ⇒suited to 1 error per statement.

Error Recovery Strategies

Phrase-level

- In this strategy the parser makes some local corrections on the remaining input on detection of an error, so that the resulting input stream give a valid construct of the language. [Example: Replacement of a , by ; or delete an extra ; or insert a missing ;]
- However, inserting a new character should be done carefully so that extraneous errors are not introduced and also the parsing algorithms can proceed without any problem.

Local correction on input

-- "," ⇒";"

-- Also decision of designer

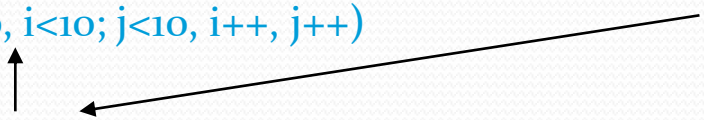
-- Not suited to all situations

-- Used in conjunction with panic mode to allow less input to be skipped

Drawbacks: It has the difficulty in coping with situations in which actual error has occurred before the point of detection.

Like

for(i=0;j=0, i<10; j<10, i++, j++)



Compiler May think this as error

Error Recovery Strategies

Error Productions:

- These involves modifying the grammar of the language to include error situations.
- In this case the compiler designer has a very good idea about the possible types of errors so that he can modify the grammar appropriately.
- The erroneous input program is thus, valid for new grammar augmented by error productions so that parser can always proceed.

Global Correction:

This is rather a theoretical approach.

- We would like compiler to make as few changes as possible in processing an incorrect input string.
- there is an **attractor**, which checks **how different tokens are from the initial inputs**, checks the closest attractor to the incorrect token is.
- This is more of a **probabilistic type of error correction**. Unfortunately, these methods are in general too costly to implement in terms of space and time.

- Given an incorrect input stream x for grammar G, find another stream y, acceptable by G, so that the number of tokens to be modified to convert x into y is the minimum.

- Approach is very costly and also not very practical.

Motivating Grammars

- **Regular Expressions**
  - → **Basis of lexical analysis**
  - → **Represent regular languages**
- **Context Free Grammars**
  - → **Basis of parsing**
  - → **Represent language constructs**
  - → **Characterize context free languages**

Context-Free Grammars

- Inherently recursive structures of a programming language are defined by a context-free grammar.

- In a context-free grammar, we have:
  - A finite set of terminals (in our case, this will be the set of tokens)$\sum$
  - A finite set of non-terminals (syntactic-variables) V_N
  - A finite set of productions rules in the following form P
    - $A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)
  - A start symbol (one of the non-terminal symbol) S

- *Where all the production are in the form* $\alpha \rightarrow \beta$ *where* $\alpha \in VN$ *, i.e., set of non-terminals and* $|\alpha| = 1$*, i.e., there will be only one non-terminal at the left hand side (LHS) and* $\beta \in VN \cup \Sigma$*, i.e.,* β *is a combination of non-terminals and terminals.*

- Example:

  $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E$

  $E \rightarrow (E)$

  $E \rightarrow id$

CFG - Terminology

- L(G) is *the language of G* (the language generated by G) which is a set of sentences.
- *A sentence of L(G)* is a string of terminal symbols of G.
- If S is the start symbol of G then

  ω is a sentence of L(G) iff $S \overset{+}{\Rightarrow} \omega$ where ω is a string of terminals of G.

- If G is a context-free grammar, L(G) is a *context-free language*.
- Two grammars are *equivalent* if they produce the same language.

- $S \overset{*}{\Rightarrow} \alpha$ - If α contains non-terminals, it is called as a *sentential* form of G.

  - If α does not contain non-terminals, it is called as a *sentence* of G.

Context Free Grammars : Concepts & Terminology

Definition: A Context Free Grammar, CFG, is described by T, NT, S, PR, where:

T: Terminals / tokens of the language

NT: Non-terminals to denote sets of strings generated by the grammar & in the language

S: Start symbol, S∈NT, which defines all strings of the language

PR: Production rules to indicate how T and NT are combined to generate valid strings of the language.

$$PR: NT \rightarrow (T \mid NT)^*$$

Like a Regular Expression / DFA / NFA, a Context Free Grammar is a mathematical model

Example : Consider a grammar to generate arithmatic expressions consisting of numbers and operator symbols +,-,*,/,and ↑.The rules of the grammar can be written as,

$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E$

$E \rightarrow (E)$

$E \rightarrow id$

We can apply the rules to derive the expression "2*(3+5*4)" as follows:

$E \rightarrow E*E \rightarrow E*(E) \rightarrow E*(E+E) \rightarrow E*(E+E*E) \rightarrow E*(E+E*4) \rightarrow E*(E+E*4) \rightarrow EA(EA5*4)$
$\rightarrow EA(E+5*4) \rightarrow EA(3+5*4) \rightarrow E*(3+5*4) \rightarrow 2*(3+5*4)$

Backus Naur Form (BNF)

- This notation was first introduced by John Backus and Peter Naur. This was first introduced for the description of the ALGOL 60 programming language.

The symbols used in BNF are as follows:

- :: = denotes “is defi ned as”
- | denotes “or”
- < > used to hold category names

As an example,

<number> :: = <digit> | <number> <digit>

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

This can be described as a number is a digit or a number followed by a digit.

- A digit is any of the characters from 0 to 9.
- From BNF comes the extended BNF which uses some notations of regular expression such as + or *.
- As an example, the identifi er of a programming language is denoted as

<Identifi er> :: = letter(letter | digit)*

<letter> :: = a | . . . . . | z | A | . . . | Z

<digit> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 BNF is widely used in CFG.

Example Grammar - Terminology

Terminals: a,b,c,+,-,punc,0,1,…,9,

Non Terminals: A,B,C,S,

T or NT: X,Y,Z

Strings of Terminals: u,v,…,z in T^*

Strings of T / NT: α, β, γ in $(T \cup NT)^*$

Alternatives of production rules:

$A \rightarrow \alpha_1; A \rightarrow \alpha_2; \ldots; A \rightarrow \alpha_k; \Rightarrow A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_1$

First NT on LHS of 1st production rule is designated as start symbol !

$E \rightarrow E\ A\ E \mid (\ E\ ) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

Example

- Expressions:

  E ::= E + T
  | E - T
  | T

  T ::= T * F
  | T / F
  | F

  F ::= num
  | id

  – … or equivalently:

  E ::= E + T
  E ::= E - T
  E ::= T
  T ::= T * F
  T ::= T / F
  T ::= F
  F ::= num
  F ::= id

- Non terminals: E T F
- Start symbol: E
- Terminals: + - * / id num
- Example: x+2*y

Derivations

- In the process of generating a language from the given production rules of a grammar, the non-terminals are replaced by the corresponding strings of the right hand side (RHS) of the production. But if there are more than one non-terminal, then which of the ones will be replaced must be determined.

$E \Rightarrow E+E$

- E+E derives from E
  - we can replace E by E+E
  - to able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$

- A sequence of replacements of non-terminal symbols is called a **derivation** of id+id from E.
- In general a derivation step is

  $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if there is a production rule $A \rightarrow \gamma$ in our grammar

  where α and β are arbitrary strings of terminal and non-terminal symbols

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

$\Rightarrow$: derives in one step

$\overset{*}{\Rightarrow}$: derives in zero or more steps

$\overset{+}{\Rightarrow}$: derives in one or more steps

Derivation Example

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

OR

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

- At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.

- **Leftmost derivation:** A derivation is called a leftmost derivation if we replace only the leftmost non-terminal by some production rule at each step of the generating process of the language from the grammar.
- **Rightmost derivation:** A derivation is called a rightmost derivation if we replace only the rightmost non-terminal by some production rule at each step of the generating process of the language from the grammar.

Left-Most and Right-Most Derivations

- S → 0S/1AA
- A → 0/1A/0B
- B → 1/0BB

Leftmost Derivation

S → 0S → 01AA → 010BA → 0100BBA → 01001BA → 010011A → 0100110

Rightmost Derivation

S → 0S → 01AA → 01A0 → 010B0 → 0100BB0 → 0100B10 → 0100110

Left-Most and Right-Most Derivations

E→ E + E/E * E/id

String id + id*id

Right Most

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

Left Most

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

Parse Tree

- Parsing a string is finding a derivation for that string from a given grammar.
- A parse tree is the tree representation of deriving a CFL from a given context grammar. These types of trees are sometimes called as derivation trees.
- *A parse tree is an ordered tree in which the LHS of a production represents a parent node and the RHS of a production represents a children node.*

Parse Tree

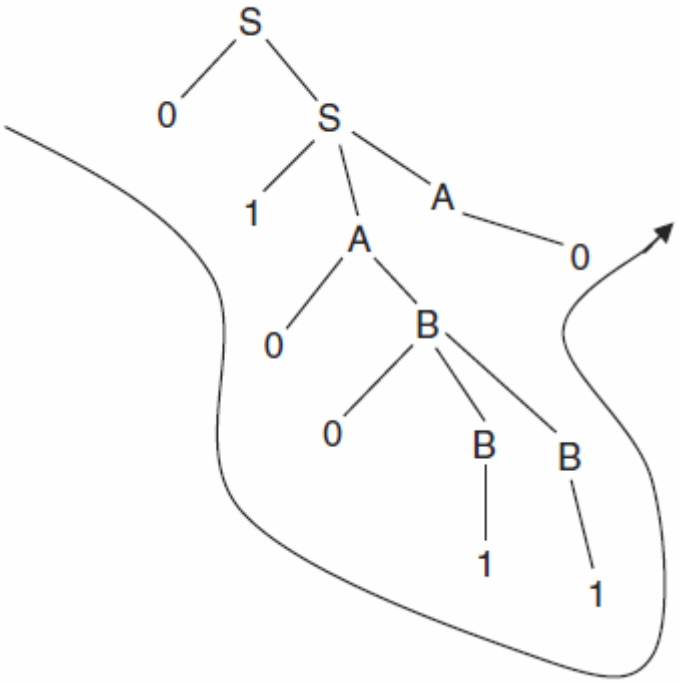
- There are certain conditions for constructing a parse tree from a given CFG. Those are as follows.

- Each vertex of the tree must have a label. The label is a non-terminal or terminal or null (Λ).
- The root of the tree is the start symbol, i.e., S.
- The label of the internal vertices is a non-terminal symbol ∈ VN.
- If there is a production $A \rightarrow X_1X_2 \ldots . X_k$, then for a vertex label A, the children of that node will be $X_1X_2 \ldots . X_K$.
- A vertex n is called a leaf of the parse tree if its label is a terminal symbol ∈ Σ or null (Λ).

Parse Tree

- The parse tree construction is possible only for CFG. This is because the properties of a tree match with the properties of CFG.
- For a tree, there must be some root. For every CFG, there is a single start symbol.
- Each node of a tree has a single label. For every CFG at the LHS, there is a single non-terminal.
- A child node is derived from a single parent. For constructing a CFL from a given CFG, a non-terminal is replaced by a suitable string at the RHS (if for a non-terminal, there are multiple productions). Each of the characters of the string is generating a node. That is, for each single node there is a single parent.

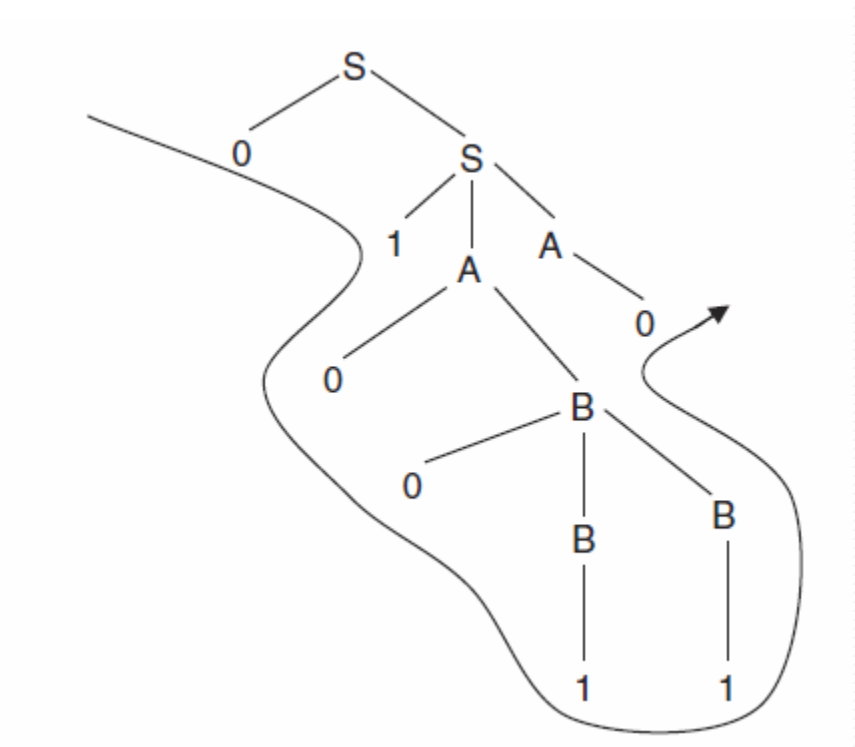
Parse Tree



S → 0S →01AA → 010BA → 0100BBA → 01001BA → 010011A → 0100110

Parse Tree for Left most derivation.

Parse Tree



Parse tree for right most derivation

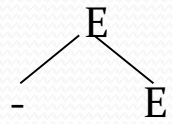
S → 0S → 01AA → 01A0 → 010B0 → 0100BB0 → 0100B10 → 0100110

Parse Tree

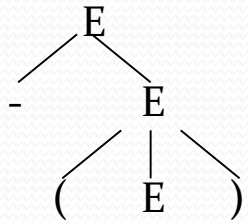
- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.

- A parse tree can be seen as a graphical representation of a derivation.

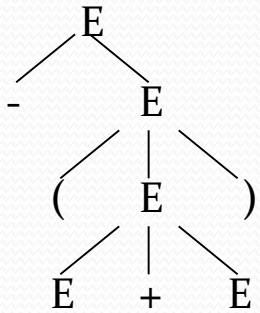
$E \Rightarrow -E$



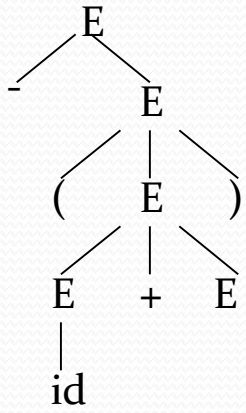
$\Rightarrow -(E)$



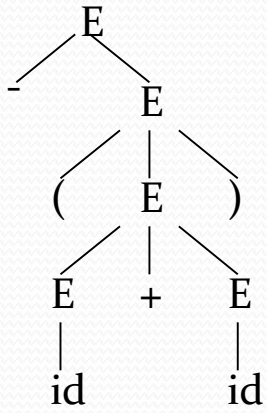
$\Rightarrow -(E+E)$



$\Rightarrow -(id+E)$



$\Rightarrow -(id+id)$



Examples of LM / RM Derivations

$E \rightarrow E\ A\ E \mid (\ E\ ) \mid -E \mid id$

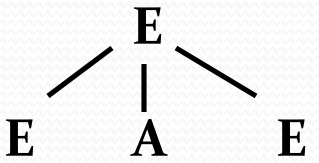
$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

A leftmost derivation of : id + id * id

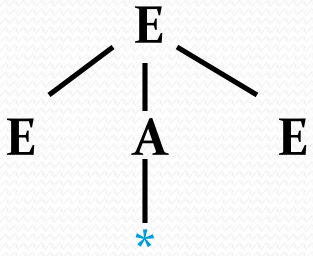
A rightmost derivation of : id + id * id

Derivations & Parse Tree

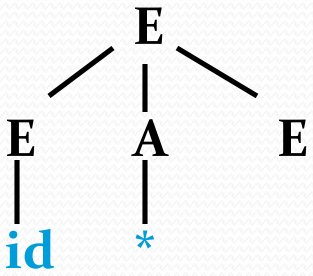
$E \Rightarrow E\ A\ E$



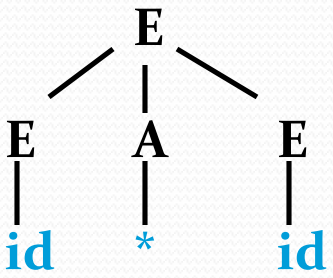
$\Rightarrow E * E$



$\Rightarrow id * E$



$\Rightarrow id * id$



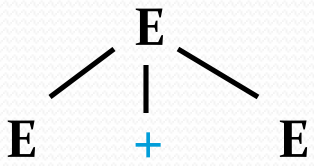
Parse Trees and Derivations

Consider the expression grammar:

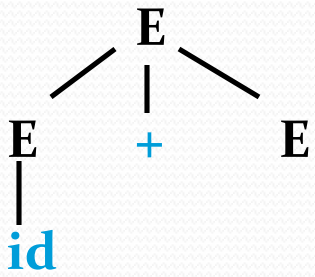
$$E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$$

Leftmost derivations of id + id * id

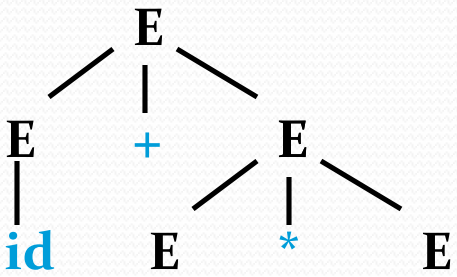
$E \Rightarrow E + E \rightarrow$



$E + E \Rightarrow id + E \rightarrow$

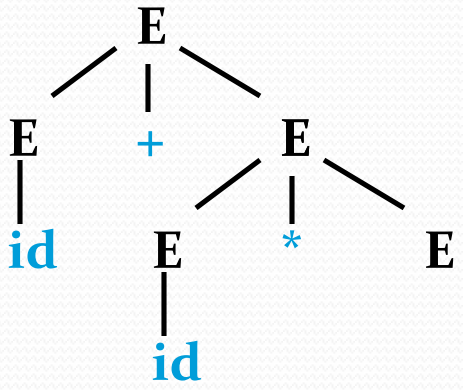


$id + E \Rightarrow id + E * E \rightarrow$

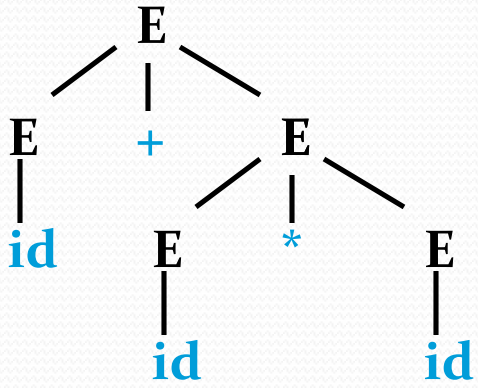


Parse Tree & Derivations - continued

$id + E * E \Rightarrow id + id * E$ →



$id + id * E \Rightarrow id + id * id$ →



Alternative Parse Tree & Derivation

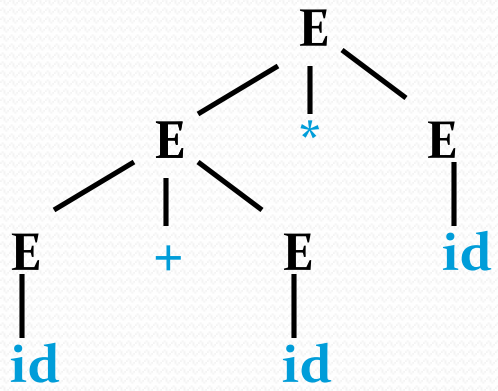
$E \Rightarrow E * E$

$\Rightarrow E + E * E$

$\Rightarrow id + E * E$

$\Rightarrow id + id * E$

$\Rightarrow id + id * id$



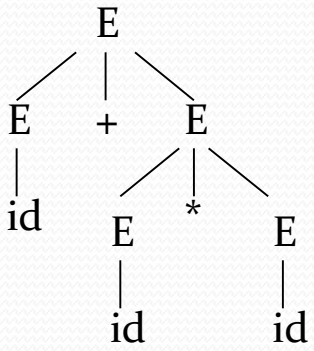
WHAT'S THE ISSUE HERE ?

Two distinct leftmost derivations!

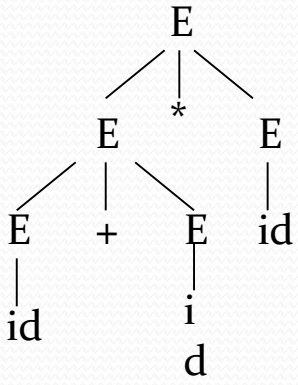
Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$
$\Rightarrow id+id*E \Rightarrow id+id*id$



$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E$
$\Rightarrow id+id*E \Rightarrow id+id*id$



Ambiguity

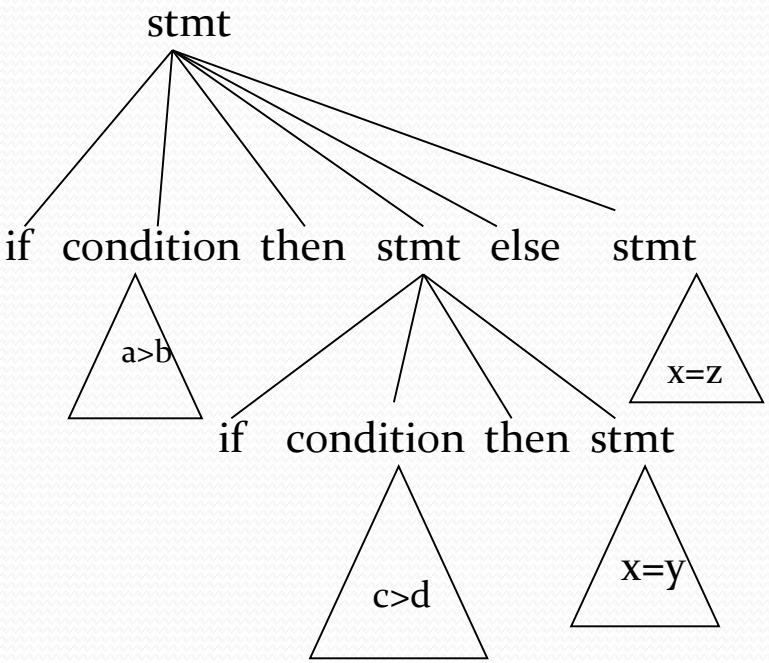
stmt → if condition then stmt else stmt

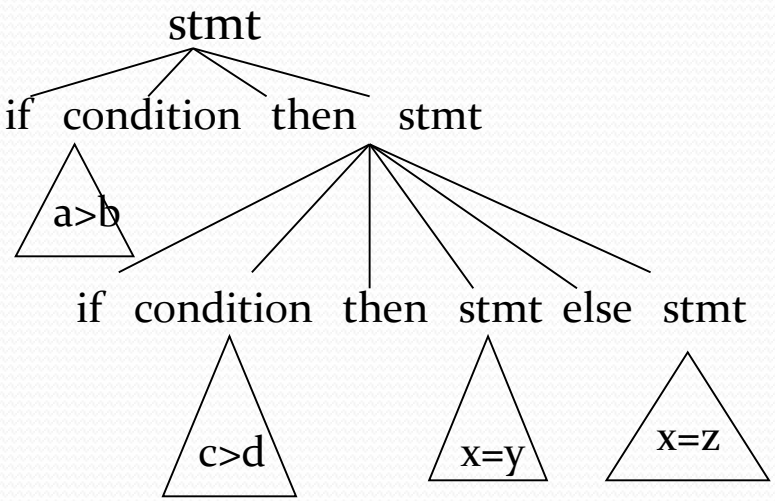
| if condition then stmt

if a>b then

if c>d then x = y

else x = z





Ambiguity (cont.)

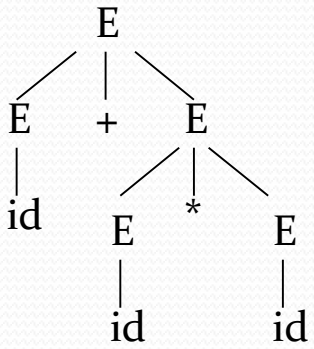
- For the most parsers, the grammar must be unambiguous.

- unambiguous grammar
  - ➔ unique selection of the parse tree for a sentence

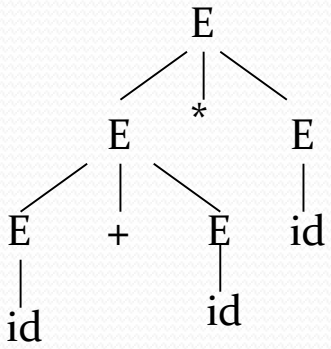
- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An unambiguous grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.

Problem in Ambiguity

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$
$\Rightarrow id+id*E \Rightarrow id+id*id$

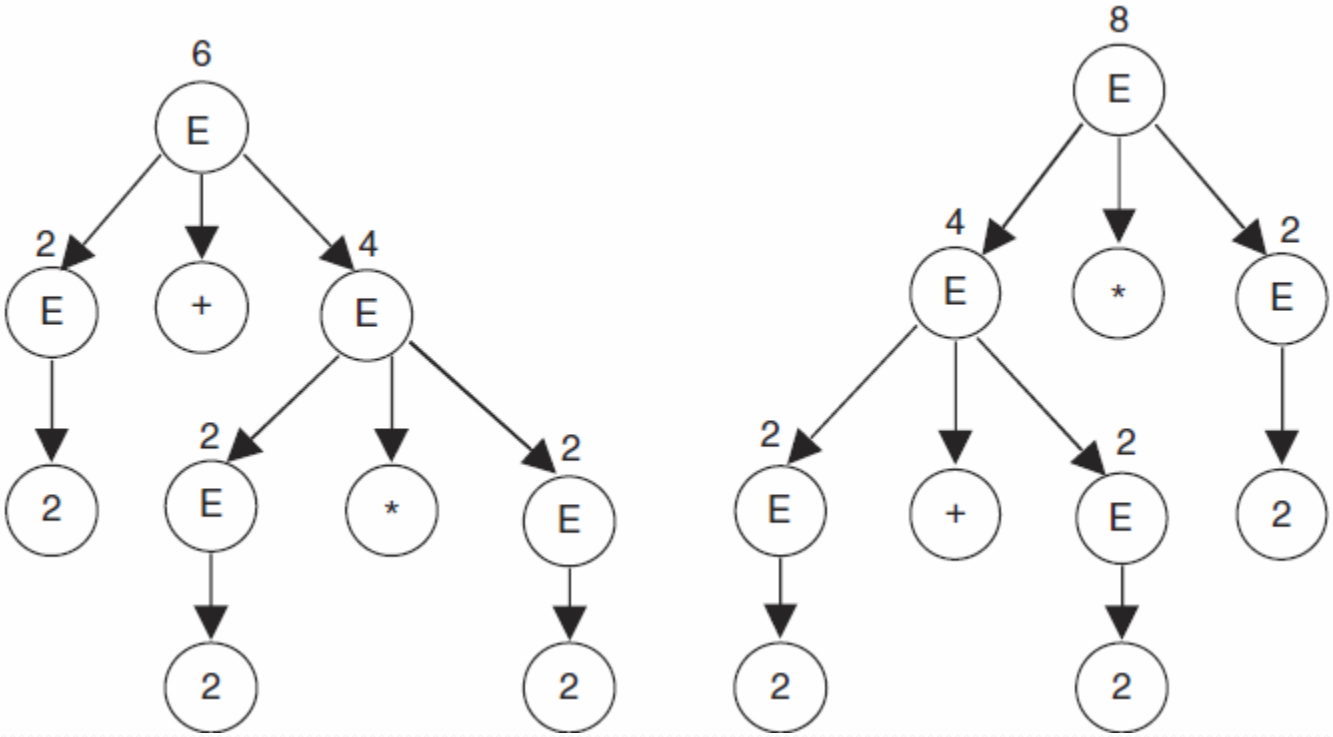


$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E$
$\Rightarrow id+id*E \Rightarrow id+id*id$



- Replace id by 2.

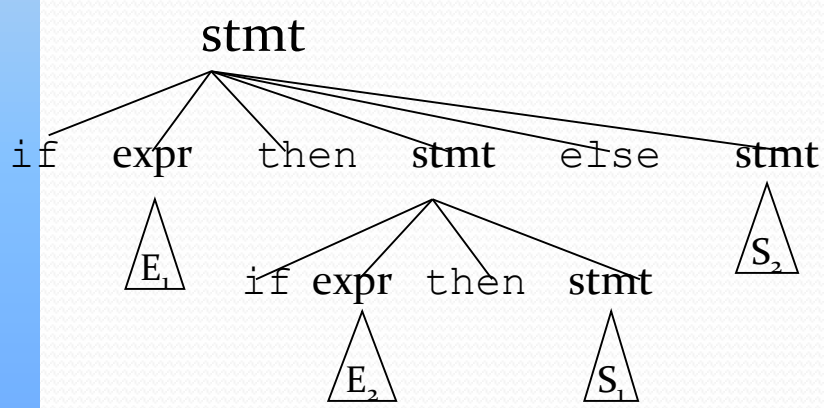
Problem in Ambiguity



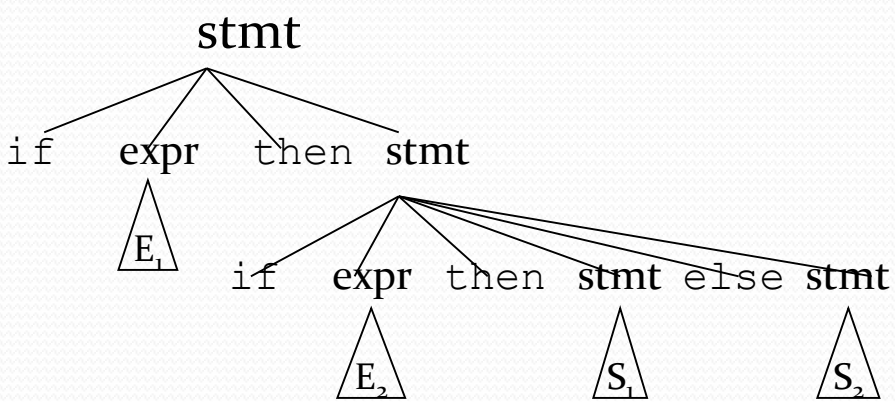
Problem in Ambiguity

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

`if` E_1 `then` `if` E_2 `then` S_1 `else` S_2



1



2

Removing Ambiguity

- There is no particular rule to remove ambiguity from a CFG. Sometimes, ambiguity can be removed by hand.
- In the previous case, ambiguity can be removed by setting priority to the operators + and *.
- If * is set a higher priority than +, then ambiguity can be removed.
- More bad news is that some CFL have only ambiguous grammar. This means, in no way the ambiguity can be removed.
- This type of ambiguity is called inherent ambiguity.

Removing Ambiguity

- take the example of if-then-else
- We prefer the second parse tree (else matches with closest if).
- So, we have to disambiguate our grammar to reflect this choice.

- To remove ambiguity following general rule is followed
  '***match each else with the closest previous unmatched then***'

- The unambiguous grammar will be:

stmt → matchedstmt | unmatchedstmt

matchedstmt → `if` expr `then` matchedstmt `else` matchedstmt | otherstmts

unmatchedstmt → `if` expr `then` stmt |
`if` expr `then` matchedstmt `else` unmatchedstmt

Ambiguity – Operator Precedence

- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the precedence and associativity rules.

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

disambiguate the grammar

precedence:

^	(right to left)
*	(left to right)
+	(left to right)

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow G\hat{}F \mid G$

$G \rightarrow id \mid (E)$

Resolving Grammar Problems/Difficulties

Regular Expressions : Basis of Lexical Analysis

Reg. Expr. → generate/represent regular languages

Reg. Languages → smallest, most well defined class of languages

Context Free Grammars: Basis of Parsing

CFGs → represent context free languages

CFLs → contain more powerful languages

Regular Expressions vs. CFGs

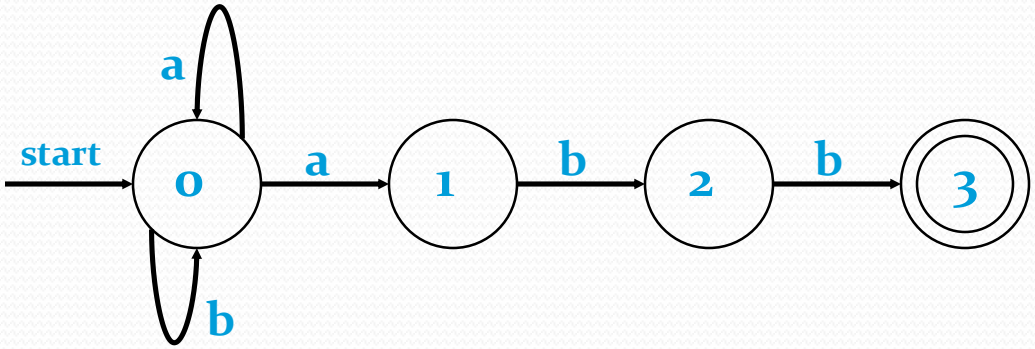
Regular expressions for lexical syntax

1. **CFGs are overkill, lexical rules are quite simple and straightforward**
2. **REs – concise / easy to understand**
3. **More efficient lexical analyzer can be constructed**
4. **RE for lexical analysis and CFGs for parsing promotes modularity, low coupling & high cohesion.**

Resolving Problems/Difficulties – (2)

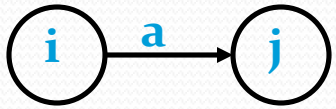
Since Reg. Lang. ⊂ Context Free Lang., it is possible to go from reg. expr. to CFGs via NFA.

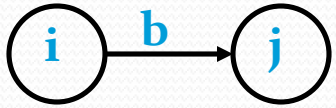
Recall: (a | b)*abb



Resolving Problems/Difficulties – (3)

Construct CFG as follows:

1. Each State I has non-terminal A_i : A_0, A_1, A_2, A_3
2. If 

then $A_i \rightarrow a\,A_j$
3. If 

then $A_i \rightarrow bA_j$

$\left.\begin{array}{l} : A_0 \rightarrow aA_0,\ A_0 \rightarrow aA_1 \\ : A_0 \rightarrow bA_0,\ A_1 \rightarrow bA_2 \\ : A_2 \rightarrow bA_3 \end{array}\right\}$

4. If I is an accepting state, $A_i \rightarrow \in$: $A_3 \rightarrow \in$
5. If I is a starting state, A_i is the start symbol : A_0

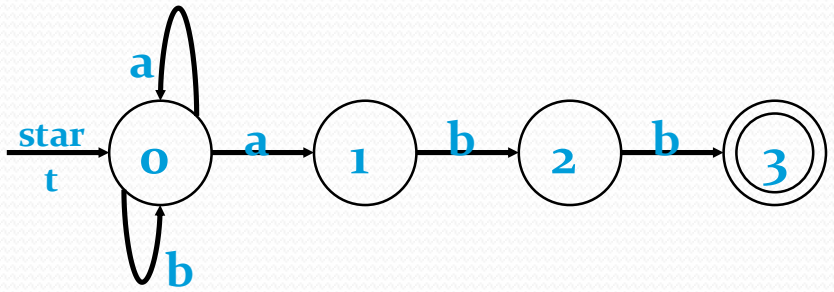
$T=\{a,b\},\ NT=\{A_0, A_1, A_2, A_3\},\ S = A_0$

$PR = \{ A_0 \rightarrow aA_0 \mid aA_1 \mid bA_0 ;$

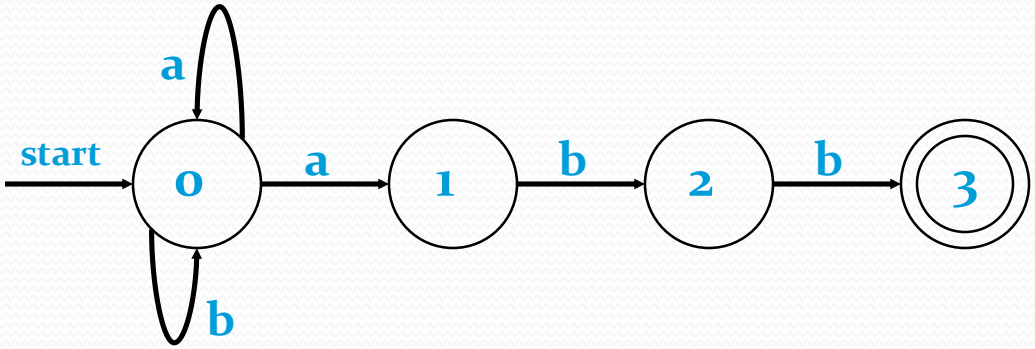
$A_1 \rightarrow bA_2 ;$

$A_2 \rightarrow bA_3 ;$

$A_3 \rightarrow \in \}$



How Does This CFG Derive Strings ?



vs.

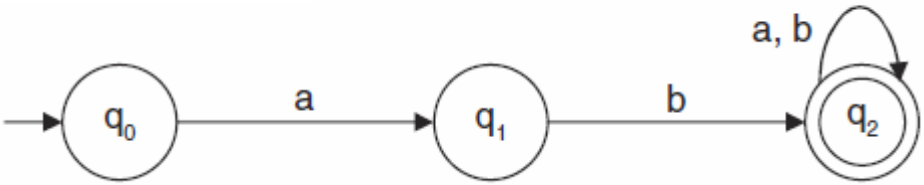
$A_0 \rightarrow aA_0, A_0 \rightarrow aA_1$

$A_0 \rightarrow bA_0, A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3, A_3 \rightarrow \in$

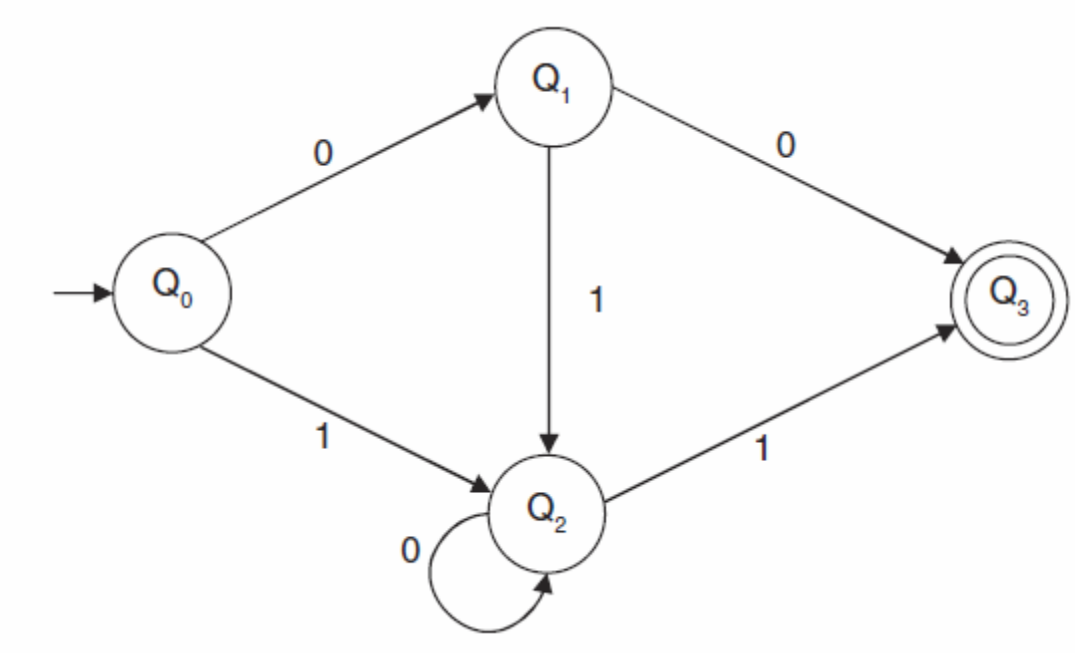
How is abaabb derived in each ?

CFG From FA



$$\{A \rightarrow aB,\ B \rightarrow bC/b,\ C \rightarrow aC/bC/a/b\}$$

CFG From FA



$\{A \rightarrow 0B/1C, B \rightarrow 0D/1C/0, C \rightarrow 0C/1D/1\}$

Left Recursion

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation.

  $A \Rightarrow^{+} A\alpha$ for some string α

- Top-down parsing techniques **cannot** handle left-recursive grammars.
- So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

Removal of Immediate Left-Recursion (Moore's Proposal)

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\,A' \mid \ldots \mid \beta_n\,A'$

$A' \rightarrow \alpha_1\,A' \mid \ldots \mid \alpha_m\,A' \mid \varepsilon$ an equivalent grammar

Immediate Left-Recursion -- Example

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

eliminate immediate left recursion

$E \rightarrow T\ E'$

$E' \rightarrow +T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *F\ T' \mid \varepsilon$

$F \rightarrow id \mid (E)$

Left-Recursion -- Problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$
$A \rightarrow Sc \mid d$

This grammar is not immediately left-recursive, but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or
$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

- So, we have to eliminate all left-recursions from our grammar

Eliminate Left-Recursion -- Algorithm

- Arrange non-terminals in some order: $A_1 \ldots A_n$
- **for** i **from** 1 **to** n **do** {
  - **for** j **from** 1 **to** i-1 **do** {
    replace each production

    $A_i \rightarrow A_j \gamma$

    by

    $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$

    where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$

    }
  - eliminate immediate left-recursions among A_i productions

}

Eliminate Left-Recursion -- Example

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$

Rename S as A1 and A as A2. The grammar is

$A_1 \rightarrow A_2\, a/b$

$A_2 \rightarrow A_1 c/d.$

For i = 1, j = 1, there is no production in the form $A1 \rightarrow A1\alpha$.

For i = 2, j = 1, there is a production in the form $A_2 \rightarrow A_1\alpha$. The production is $A_2 \rightarrow A_1 c$.
According to the algorithm for removal of indirect left recursion, the production becomes

$A_2 \rightarrow A_2 ac/bc$

This left recursion for the production $A_2 \rightarrow A_2 ac/bc/d$ is removed and the production rules are

- $A_2 \rightarrow bcA_2'\ /dA_2'$
- $A_2' \rightarrow acA_2'\ /\in$

The actual non-left recursive grammar is

$S \rightarrow Aa/b$

$A \rightarrow bcA'/dA'$

$A' \rightarrow acA'/\in.$

Eliminate Left-Recursion -- Example

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid f$

Solution: The grammar has indirect left recursion. Rename S as A1 and A as A2. The grammar is

$A_1 \rightarrow A_2a/b$

$A_2 \rightarrow A_2c/A_1d/f$

For i = 1, j = 1, there is no production in the form $A_1 \rightarrow A_1\alpha$.

For i = 2, j = 1, there is a production in the form $A_2 \rightarrow A_1\alpha$. The production is $A_2 \rightarrow A_1d$. According to the algorithm for removal of indirect left recursion, the production becomes

$A_2 \rightarrow A_2ad/bd$

The A_2 production is $A_2 \rightarrow A_2c/A_2ad/bd/f$.

Eliminate Left-Recursion -- Example

- This left recursion for the production $A_2 \rightarrow A_2c/A_2ad/bd/f$ is removed and the production rules are

$A_2 \rightarrow bdA_2'$

$A_2 \rightarrow fA_2'$

$A'_2 \rightarrow cA_2'/adA_2'/\varepsilon$

The actual non-left recursive grammar is

$S \rightarrow Aa \mid b$

$A \rightarrow bdA'/fA'$

$A' \rightarrow cA'/adA'/\varepsilon$

Assignment

S→ (L)/a
L→LS/S

Remove Left recursion from the given grammar.
A→Bx/ Cx/y
B → Ay/ε
C → Ax/y

Remove Left recursion from the given grammar.

S→ BC
A→ SC
B→ CA/b
C →AB/a

Left-Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

  grammar ➔ a new equivalent grammar suitable for predictive parsing

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

- when we see `if`, we cannot now which production rule to choose to write *stmt* in the derivation.

Left-Factoring (cont.)

- In general,

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different.

- when processing α we cannot know whether expand

  A to $\alpha\beta_1$ or

  A to $\alpha\beta_2$

- But, if we re-write the grammar as follows

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left-Factoring -- Algorithm

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

Left-Factoring – Example1

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$A \rightarrow aA' \mid \underline{cdg} \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Left-Factoring – Example2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$