Lecture 6

Syntax Analysis

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

January 31, 2025

Take aways from the last class

- Extended regular expressions

Take aways from the last class

- Extended regular expressions
- Lexical Analyzer generator

Take aways from the last class

- Extended regular expressions
- Lexical Analyzer generator
- Lex file format and compilation steps

Take aways from the last class

- Extended regular expressions
- Lexical Analyzer generator
- Lex file format and compilation steps
- Working principle of the `lex`

Take aways from the last class

- Extended regular expressions
- Lexical Analyzer generator
- Lex file format and compilation steps
- Working principle of the `lex`
- Correctness check of a string based on `lex` rules

move from one transition diagram to the next diagram.

Take aways from the last class

- Extended regular expressions
- Lexical Analyzer generator
- Lex file format and compilation steps
- Working principle of the `lex`
- Correctness check of a string based on `lex` rules
- Interface with other passes

Overview of Syntax Analysis

- Check syntax and construct abstract syntax tree

Overview of Syntax Analysis

- Check syntax and construct abstract syntax tree
- Error reporting and recovery

Overview of Syntax Analysis

- Check syntax and construct abstract syntax tree
- Error reporting and recovery
- Model using context-free grammars

Overview of Syntax Analysis

- Check syntax and construct abstract syntax tree
- Error reporting and recovery
- Model using context-free grammars
- Recognize using push-down automata/table-driven Parsers

Overview of Syntax Analysis

- Check syntax and construct abstract syntax tree
- Error reporting and recovery
- Model using context-free grammars
- Recognize using push-down automata/table-driven Parsers

- To check whether variables are of types on which operations are allowed

Overview of Syntax Analysis

- Check syntax and construct abstract syntax tree
- Error reporting and recovery
- Model using context-free grammars
- Recognize using push-down automata/table-driven Parsers
- To check whether variables are of types on which operations are allowed ✗

Overview of Syntax Analysis

- Check syntax and construct abstract syntax tree
- Error reporting and recovery
- Model using context-free grammars
- Recognize using push-down automata/table-driven Parsers
- To check whether variables are of types on which operations are allowed ✗
- To check whether a variable has been declared before use

Overview of Syntax Analysis

- Check syntax and construct abstract syntax tree
- Error reporting and recovery
- Model using context-free grammars
- Recognize using push-down automata/table-driven Parsers
- To check whether variables are of types on which operations are allowed ✗
- To check whether a variable has been declared before use ✗

Overview of Syntax Analysis

- Check syntax and construct abstract syntax tree
- Error reporting and recovery
- Model using context-free grammars
- Recognize using push-down automata/table-driven Parsers
- To check whether variables are of types on which operations are allowed ✗
- To check whether a variable has been declared before use ✗
- To check whether a variable has been initialized

Overview of Syntax Analysis

- Check syntax and construct abstract syntax tree
- Error reporting and recovery
- Model using context-free grammars
- Recognize using push-down automata/table-driven Parsers
- To check whether variables are of types on which operations are allowed ✗
- To check whether a variable has been declared before use ✗
- To check whether a variable has been initialized ✗

Overview of Syntax Analysis

- Check syntax and construct abstract syntax tree
- Error reporting and recovery
- Model using context-free grammars
- Recognize using push-down automata/table-driven Parsers
- To check whether variables are of types on which operations are allowed ✗
- To check whether a variable has been declared before use ✗
- To check whether a variable has been initialized ✗
- These issues will be handled in semantic analysis

Derivation

Does $9-5+2$ belong to the following grammar?

$$
\begin{aligned}
list &\rightarrow list + digit\\
&\quad |list - digit\\
&\quad |digit\\
digit &\rightarrow 0|1|2\ldots|9
\end{aligned}
$$

Derivation

Does $9 - 5 + 2$ belong to the following grammar?

$$
\begin{aligned}
list &\rightarrow list + digit \\
&\quad |list - digit \\
&\quad |digit \\
digit &\rightarrow 0|1|2 \ldots |9
\end{aligned}
$$

String Derivation:

Derivation

Does $9 - 5 + 2$ belong to the following grammar?

$$
\begin{aligned}
\mathit{list} &\rightarrow \mathit{list} + \mathit{digit} \\
&\quad |\mathit{list} - \mathit{digit} \\
&\quad |\mathit{digit} \\
\mathit{digit} &\rightarrow 0|1|2\ldots|9
\end{aligned}
$$

String Derivation:

$\mathit{list} \rightarrow \underline{\mathit{list}} + \mathit{digit}$

Derivation

Does $9 - 5 + 2$ belong to the following grammar?

$$\begin{aligned} \textit{list} &\rightarrow \textit{list} + \textit{digit} \\ &\quad |\textit{list} - \textit{digit} \\ &\quad |\textit{digit} \\ \textit{digit} &\rightarrow 0|1|2\ldots|9 \end{aligned}$$

String Derivation:

$\textit{list} \rightarrow \underline{\textit{list}} + \textit{digit}$

$\textit{list} \rightarrow \underline{\textit{list}} - \textit{digit} + \textit{digit}$

Derivation

Does $9 - 5 + 2$ belong to the following grammar?

$$\begin{aligned} \mathit{list} &\rightarrow \mathit{list} + \mathit{digit} \\ &|\mathit{list} - \mathit{digit} \\ &|\mathit{digit} \end{aligned}$$

$\mathit{digit} \rightarrow 0|1|2\ldots|9$

String Derivation:

$\mathit{list} \rightarrow \underline{\mathit{list}} + \mathit{digit}$

$\mathit{list} \rightarrow \underline{\mathit{list}} - \mathit{digit} + \mathit{digit}$

$\mathit{list} \rightarrow \underline{\mathit{digit}} - \mathit{digit} + \mathit{digit}$

Derivation

Does $9 - 5 + 2$ belong to the following grammar?

$$
\begin{aligned}
\mathit{list} &\rightarrow \mathit{list} + \mathit{digit} \\
&\quad |\mathit{list} - \mathit{digit} \\
&\quad |\mathit{digit} \\
\mathit{digit} &\rightarrow 0|1|2\ldots|9
\end{aligned}
$$

String Derivation:

$$\mathit{list} \rightarrow \underline{\mathit{list}} + \mathit{digit}$$

$$\mathit{list} \rightarrow \underline{\mathit{list}} - \mathit{digit} + \mathit{digit}$$

$$\mathit{list} \rightarrow \underline{\mathit{digit}} - \mathit{digit} + \mathit{digit}$$

$$\mathit{list} \rightarrow 9 - \underline{\mathit{digit}} + \mathit{digit}$$

Derivation

Does $9 - 5 + 2$ belong to the following grammar?

$$\begin{aligned} \textit{list} &\rightarrow \textit{list} + \textit{digit} \\ &\quad |\textit{list} - \textit{digit} \\ &\quad |\textit{digit} \end{aligned}$$

$\textit{digit} \rightarrow 0|1|2\ldots|9$

String Derivation:

$\textit{list} \rightarrow \underline{\textit{list}} + \textit{digit}$

$\textit{list} \rightarrow \underline{\textit{list}} - \textit{digit} + \textit{digit}$

$\textit{list} \rightarrow \underline{\textit{digit}} - \textit{digit} + \textit{digit}$

$\textit{list} \rightarrow 9 - \underline{\textit{digit}} + \textit{digit}$

$\textit{list} \rightarrow 9 - 5 + \underline{\textit{digit}}$

Derivation

Does $9 - 5 + 2$ belong to the following grammar?

$$\begin{aligned} \mathit{list} \rightarrow\ & \mathit{list} + \mathit{digit} \\ & |\mathit{list} - \mathit{digit} \\ & |\mathit{digit} \end{aligned}$$

$\mathit{digit} \rightarrow 0|1|2\ldots|9$

String Derivation:

$\mathit{list} \rightarrow \underline{\mathit{list}} + \mathit{digit}$

$\mathit{list} \rightarrow \underline{\mathit{list}} - \mathit{digit} + \mathit{digit}$

$\mathit{list} \rightarrow \underline{\mathit{digit}} - \mathit{digit} + \mathit{digit}$

$\mathit{list} \rightarrow 9 - \underline{\mathit{digit}} + \mathit{digit}$

$\mathit{list} \rightarrow 9 - 5 + \underline{\mathit{digit}}$

$\mathit{list} \rightarrow 9 - 5 + 2$

Derivation

Does $9 - 5 + 2$ belong to the following grammar?

$$\begin{aligned} \mathit{list} &\rightarrow \mathit{list} + \mathit{digit} \\ &|\mathit{list} - \mathit{digit} \\ &|\mathit{digit} \end{aligned}$$

$\mathit{digit} \rightarrow 0|1|2\ldots|9$

String Derivation:

$$\begin{aligned} \mathit{list} &\rightarrow \underline{\mathit{list}} + \mathit{digit} \\ \mathit{list} &\rightarrow \underline{\mathit{list}} - \mathit{digit} + \mathit{digit} \\ \mathit{list} &\rightarrow \underline{\mathit{digit}} - \mathit{digit} + \mathit{digit} \\ \mathit{list} &\rightarrow 9 - \underline{\mathit{digit}} + \mathit{digit} \\ \mathit{list} &\rightarrow 9 - 5 + \underline{\mathit{digit}} \\ \mathit{list} &\rightarrow 9 - 5 + 2 \end{aligned}$$

- Which non-terminal should I choose?

Derivation

Does $9 - 5 + 2$ belong to the following grammar?

$$
\begin{aligned}
list &\rightarrow list + digit \\
&\quad |list - digit \\
&\quad |digit \\
digit &\rightarrow 0|1|2\ldots|9
\end{aligned}
$$

String Derivation:

$$
\begin{aligned}
list &\rightarrow \underline{list} + digit \\
list &\rightarrow \underline{list} - digit + digit \\
list &\rightarrow \underline{digit} - digit + digit \\
list &\rightarrow 9 - \underline{digit} + digit \\
list &\rightarrow 9 - 5 + \underline{digit} \\
list &\rightarrow 9 - 5 + 2
\end{aligned}
$$

- Which non-terminal should I choose?
- Which production rule should I select?

Derivation

- If there is a production $A \rightarrow \alpha$ then we say that A derives α and is denoted by $A \Rightarrow \alpha$

Derivation

- If there is a production $A \rightarrow \alpha$ then we say that A derives α and is denoted by $A \Rightarrow \alpha$
- $\alpha A\beta \Rightarrow \alpha\gamma\beta$ if $A \rightarrow \gamma$ is a production

Derivation

- If there is a production $A \rightarrow \alpha$ then we say that A derives α and is denoted by $A \Rightarrow \alpha$
- $\alpha A\beta \Rightarrow \alpha\gamma\beta$ if $A \rightarrow \gamma$ is a production
- If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ then $\alpha_1 \Rightarrow^+ \alpha_n$

Derivation

- If there is a production $A \rightarrow \alpha$ then we say that A derives α and is denoted by $A \Rightarrow \alpha$
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production
- If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ then $\alpha_1 \Rightarrow^+ \alpha_n$
- If $S \Rightarrow^+ \alpha$ where α is a string of terminals and non-terminals of G then we say that α is a **sentential** form of G.

sentential form may have non-terminals as well as terminals.

Derivation

- If there is a production $A \rightarrow \alpha$ then we say that A derives α and is denoted by $A \Rightarrow \alpha$
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production
- If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ then $\alpha_1 \Rightarrow^+ \alpha_n$
- If $S \Rightarrow^+ \alpha$ where α is a string of terminals and non-terminals of G then we say that α is a **sentential** form of G.
- If in a sentential form only the leftmost non terminal is replaced then it becomes leftmost derivation

Derivation

- If there is a production $A \rightarrow \alpha$ then we say that A derives α and is denoted by $A \Rightarrow \alpha$
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production
- If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ then $\alpha_1 \Rightarrow^+ \alpha_n$
- If $S \Rightarrow^+ \alpha$ where α is a string of terminals and non-terminals of G then we say that α is a **sentential** form of G.
- If in a sentential form only the leftmost non terminal is replaced then it becomes leftmost derivation
- Every leftmost step can be written as $wA\gamma \Rightarrow^{lm*} w\delta\gamma$ where w is a string of terminals and $A \rightarrow \delta$ is a production.

Derivation

- If there is a production $A \rightarrow \alpha$ then we say that A derives α and is denoted by $A \Rightarrow \alpha$
- $\alpha A\beta \Rightarrow \alpha\gamma\beta$ if $A \rightarrow \gamma$ is a production
- If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ then $\alpha_1 \Rightarrow^+ \alpha_n$
- If $S \Rightarrow^+ \alpha$ where α is a string of terminals and non-terminals of G then we say that α is a **sentential** form of G.
- If in a sentential form only the leftmost non terminal is replaced then it becomes leftmost derivation
- Every leftmost step can be written as $wA\gamma \Rightarrow^{lm*} w\delta\gamma$ where w is a string of terminals and $A \rightarrow \delta$ is a production.
- Similarly, right most derivation can be defined.

Derivation

do note the meaning of double and single arrow...

- If there is a production $A \rightarrow \alpha$ then we say that A derives α and is denoted by $A \Rightarrow \alpha$
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production
- If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ then $\alpha_1 \Rightarrow^+ \alpha_n$
- If $S \Rightarrow^+ \alpha$ where α is a string of terminals and non-terminals of G then we say that α is a **sentential** form of G.
- If in a sentential form only the leftmost non terminal is replaced then it becomes leftmost derivation
- Every leftmost step can be written as $wA\gamma \Rightarrow^{lm*} w\delta\gamma$ where w is a string of terminals and $A \rightarrow \delta$ is a production.
- Similarly, right most derivation can be defined.
- An ambiguous grammar is one that produces more than one leftmost/rightmost derivation of a sentence

Parse Tree

- It shows how the start symbol of a grammar derives a string in the language.

Parse Tree

- It shows how the start symbol of a grammar derives a string in the language.
- *root* is labeled by the start symbol

Parse Tree

- It shows how the start symbol of a grammar derives a string in the language.
- *root* is labeled by the start symbol
- *leaf* nodes are labeled by tokens

Parse Tree

- It shows how the start symbol of a grammar derives a string in the language.
- *root* is labeled by the start symbol
- *leaf* nodes are labeled by tokens
- Each internal node is labeled by a non-terminal

Parse Tree

- It shows how the start symbol of a grammar derives a string in the language.
- *root* is labeled by the start symbol
- *leaf* nodes are labeled by tokens
- Each internal node is labeled by a non-terminal
- If A is a non-terminal labeling an internal node and $x_1, x_2, \ldots x_n$ are labels of the children of that node, then $A \rightarrow x_1 x_2 \ldots x_n$ is a production

Ambiguity

Ambiguity

- A Grammar can have more than one parse tree for a string

Ambiguity

- A Grammar can have more than one parse tree for a string
- Consider grammar

Ambiguity

- A Grammar can have more than one parse tree for a string
- Consider grammar

$$
\begin{aligned}
string \rightarrow\ & string + string \\
& | string - string \\
& | 0 | 1 | \cdots | 9
\end{aligned}
$$

Ambiguity

- A Grammar can have more than one parse tree for a string
- Consider grammar

  $$\begin{aligned} \mathit{string} &\rightarrow \mathit{string} + \mathit{string} \\ &\quad|\mathit{string} - \mathit{string} \\ &\quad|0|1|\cdots|9 \end{aligned}$$

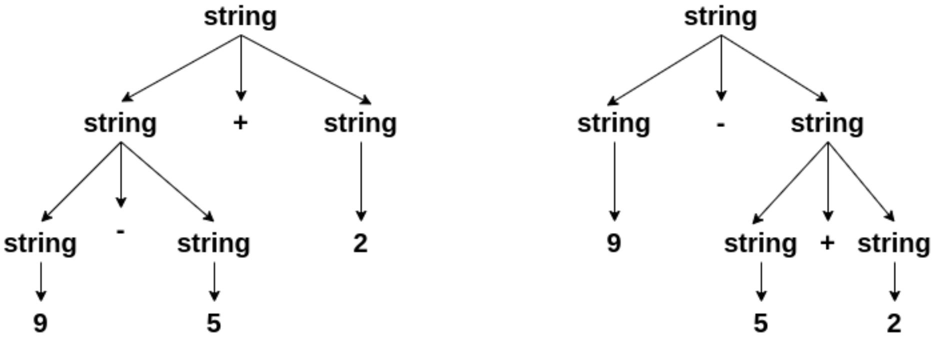
- String $9 - 5 + 2$ has two parse trees

Ambiguity

- A Grammar can have more than one parse tree for a string
- Consider grammar

  $$\begin{aligned} string &\rightarrow string + string \\ &\quad | string - string \\ &\quad |0|1|\cdots|9 \end{aligned}$$

- String $9 - 5 + 2$ has two parse trees



Ambiguity

- Ambiguity is problematic because meaning of the programs can be incorrect

Ambiguity

- Ambiguity is problematic because meaning of the programs can be incorrect
- Ambiguity can be handled in several ways

Ambiguity

- Ambiguity is problematic because meaning of the programs can be incorrect
- Ambiguity can be handled in several ways
  - Enforce associativity and precedence

Ambiguity

- Ambiguity is problematic because meaning of the programs can be incorrect
- Ambiguity can be handled in several ways
  - Enforce associativity and precedence
  - Rewrite the grammar (cleanest way)

Ambiguity

- Ambiguity is problematic because meaning of the programs can be incorrect
- Ambiguity can be handled in several ways
  - Enforce associativity and precedence
  - Rewrite the grammar (cleanest way)
- There are no general techniques for handling ambiguity

Ambiguity

- Ambiguity is problematic because meaning of the programs can be incorrect
- Ambiguity can be handled in several ways
  - Enforce associativity and precedence
  - Rewrite the grammar (cleanest way)
- There are no general techniques for handling ambiguity
- It is impossible to convert automatically an ambiguous grammar to an unambiguous one.

in programming

Associativity and Precedence

- If an operand has operators on both of the sides, the side on which operators takes this operand is the associativity of that operator.

Associativity and Precedence

- If an operand has operators on both of the sides, the side on which operators takes this operand is the associativity of that operator.
- In $a + b + c$ b is taken by left $+$

Associativity and Precedence

- If an operand has operators on both of the sides, the side on which operators takes this operand is the associativity of that operator.
- In $a + b + c$ b is taken by left $+$
- $+, -, *, /$ are left associative

Associativity and Precedence

- If an operand has operators on both of the sides, the side on which operators takes this operand is the associativity of that operator.
- In $a + b + c$ b is taken by left $+$
- $+, -, *, /$ are left associative
- $\hat{} =$ are right associative

Associativity and Precedence

- If an operand has operators on both of the sides, the side on which operators takes this operand is the associativity of that operator.
- In $a + b + c$ b is taken by left $+$
- $+, -, *, /$ are left associative
- ˆ$=$ are right associative
- String a+5*2 has two possible interpretations because of two different parse trees corresponding to $(a + 5) * 2$ and $a + (5 * 2)$.

Associativity and Precedence

- If an operand has operators on both of the sides, the side on which operators takes this operand is the associativity of that operator.
- In $a + b + c$ b is taken by left $+$
- $+, -, *, /$ are left associative
- $\hat{} =$ are right associative
- String a+5*2 has two possible interpretations because of two different parse trees corresponding to $(a + 5) * 2$ and $a + (5 * 2)$.
- Precedence determines the correct interpretation.

a = b = c means first assign the value of c to b and then b to a.

Ambiguity

- Dangling else problem

Ambiguity

- Dangling else problem

$$Stmt \rightarrow if \quad expr \quad then \quad stmt$$
$$\quad\quad | if \quad expr \quad then \quad stmt \quad else \quad stmt$$

Ambiguity

- Dangling else problem
  $Stmt \rightarrow if \quad expr \quad then \quad stmt$
  $\qquad\quad |if \quad expr \quad then \quad stmt \quad else \quad stmt$
- `if el then if e2 then S1 else S2` has two parse trees

Ambiguity

- Dangling else problem
  $Stmt \rightarrow if \quad expr \quad then \quad stmt$
  $\quad\quad\quad |if \quad expr \quad then \quad stmt \quad else \quad stmt$
- `if el then if e2 then S1 else S2` has two parse trees

```
if(e1)
    if(e2)
        S1
    else
        S2
```

Ambiguity

- Dangling else problem
  $Stmt \rightarrow if \quad expr \quad then \quad stmt$
  $\quad\quad |if \quad expr \quad then \quad stmt \quad else \quad stmt$
- `if el then if e2 then S1 else S2` has two parse trees

```
if(e1)
    if(e2)
        S1
    else
        S2
```

```
if(e1)
    if(e2)
        S1
else
    S2
```

Resolving dangling `else` problem

- Match each `else` with the closest previous `then`

Resolving dangling `else` problem

- Match each `else` with the closest previous `then`
- $\mathit{stmt} \rightarrow$ `matched-stmt`
  $|$ `unmatched-stmt`

Resolving dangling `else` problem

- Match each `else` with the closest previous `then`
- $\textit{stmt} \rightarrow$ `matched-stmt`
  $\mid$ `unmatched-stmt`
- `matched-stmt` $\rightarrow$ `if expr then matched-stmt else matched-stmt`
  $\mid \textit{others}$

Resolving dangling `else` problem

to resolve ambiguity, we want some disambiguating rules such as this one

- Match each `else` with the closest previous `then`
- *stmt* $\rightarrow$ `matched-stmt`
  $\mid$ `unmatched-stmt`
- `matched-stmt` $\rightarrow$ `if expr then matched-stmt else matched-stmt`
  $\mid$ *others*
- `unmatched-stmt` $\rightarrow$ `if expr then stmt`
  $\mid$ `if expr then matched-stmt else unmatched-stmt`

Parsing

- Process of determination whether a string can be generated by a grammar.

Parsing

- Process of determination whether a string can be generated by a grammar.
- Parsing falls in two categories:

Parsing

- Process of determination whether a string can be generated by a grammar.
- Parsing falls in two categories:
    - **Top-down parsing:** Construction of the parse tree starts at the root (from the start symbol) and proceeds towards leaves (token or terminals). Ex ANTLR

Parsing

- Process of determination whether a string can be generated by a grammar.
- Parsing falls in two categories:
  - **Top-down parsing:** Construction of the parse tree starts at the root (from the start symbol) and proceeds towards leaves (token or terminals). Ex ANTLR
  - **Bottom-up parsing:** Construction of the parse tree starts from the leaf nodes (tokens or terminals of the grammar) and proceeds towards root (start symbol). Ex YACC and BISON

Top Down Parsing

- Construction of a parse tree is done by starting the root labeled by a start symbol

Top Down Parsing

- Construction of a parse tree is done by starting the root labeled by a start symbol
- repeat following two steps

Top Down Parsing

- Construction of a parse tree is done by starting the root labeled by a start symbol
- repeat following two steps
  - at a node labeled with non terminal A select one of the productions of A and construct children nodes (Which production?)

Top Down Parsing

- Construction of a parse tree is done by starting the root labeled by a start symbol
- repeat following two steps
  - at a node labeled with non terminal A select one of the productions of A and construct children nodes (Which production?)
  - find the next node at which subtree is Constructed (Which node?)

Top Down Parsing

- Construction of a parse tree is done by starting the root labeled by a start symbol
- repeat following two steps
  - at a node labeled with non terminal A select one of the productions of A and construct children nodes (Which production?)
  - find the next node at which subtree is Constructed (Which node?)

Recursive Descent parsing

for every non-terminal, associate a recusive procedure to it.

Algorithm A()

Choose an A-production, $A \rightarrow X_1 X_2 \cdots X_k$
for $i = 1$ to k **do**
    if X_i is a nonterminal **then**
        call procedure$X_i()$
    else if X_i equals the current input symbol α **then**
        advance the input to the next symbol
    else
        error()
    end if
end for

Recursive Descent parsing

- Non-deterministic due to line 1 of the Algorithm 1

Recursive Descent parsing

- Non-deterministic due to line 1 of the Algorithm 1
- Require backtracking

Recursive Descent parsing

- Non-deterministic due to line 1 of the Algorithm 1
- Require backtracking
- May require repeated scans over the input.

Recursive Descent parsing

we are choosing production randomly, and it required backtracking over all the productions at any stage.

- Non-deterministic due to line 1 of the Algorithm 1
- Require backtracking
- May require repeated scans over the input.
- Dynamic Programming or tabular method may be used.

think like backtracking is itself a non-deterministic algo.

note that using backtracking will not remove non-determinism in procedure, as we are not sure of using what production.

Left Recursion

- A top-down parser with production $A \to A\alpha$ may loop forever.

Left Recursion

- A top-down parser with production $A \rightarrow A\alpha$ may loop forever.
- From the grammar $A \rightarrow A\alpha|\beta$ left recursion may be eliminated by transforming the grammar to

Left Recursion

- A top-down parser with production $A \rightarrow A\alpha$ may loop forever.
- From the grammar $A \rightarrow A\alpha|\beta$ left recursion may be eliminated by transforming the grammar to

  $A \rightarrow \beta R$

Left Recursion

- A top-down parser with production $A \rightarrow A\alpha$ may loop forever.
- From the grammar $A \rightarrow A\alpha|\beta$ left recursion may be eliminated by transforming the grammar to

  $A \rightarrow \beta R$

  $R \rightarrow \alpha R|\epsilon$

Left Recursion

- A top-down parser with production $A \rightarrow A\alpha$ may loop forever.
- From the grammar $A \rightarrow A\alpha|\beta$ left recursion may be eliminated by transforming the grammar to

  $A \rightarrow \beta R$

  $R \rightarrow \alpha R|\epsilon$
- In general $A \rightarrow A\alpha_1|A\alpha_2|\cdots|A\alpha_m|\beta_1|\beta_2|\cdots|\beta_n$ transforms to

Left Recursion

- A top-down parser with production $A \rightarrow A\alpha$ may loop forever.
- From the grammar $A \rightarrow A\alpha|\beta$ left recursion may be eliminated by transforming the grammar to

  $A \rightarrow \beta R$

  $R \rightarrow \alpha R|\epsilon$
- In general $A \rightarrow A\alpha_1|A\alpha_2|\cdots|A\alpha_m|\beta_1|\beta_2|\cdots|\beta_n$ transforms to

  $A \rightarrow \beta_1 A'|\beta_2 A'|\cdots|\beta_n A'$

  $A' \rightarrow \alpha_1 A'|\alpha_2 A'|\cdots|\alpha_m A'|\epsilon$

Example

- Consider grammar for arithmetic expressions

$$E \rightarrow E + T | T$$
$$T \rightarrow T * F | F$$
$$F \rightarrow (E) | id$$

Example

- Consider grammar for arithmetic expressions

$$E \rightarrow E + T | T$$
$$T \rightarrow T * F | F$$
$$F \rightarrow (E) | id$$

- After removal of left recursion the grammar becomes

Example

- Consider grammar for arithmetic expressions

$$E \rightarrow E + T | T$$
$$T \rightarrow T * F | F$$
$$F \rightarrow (E) | id$$

- After removal of left recursion the grammar becomes

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' | \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' | \epsilon$$
$$F \rightarrow (E) | id$$

Left recursion hidden due to many productions

- Left recursion may also be introduced by two or more grammar rules. For example:
  $S \to Aa|b$
  $A \to Ac|Sd|\epsilon$

Left recursion hidden due to many productions

- Left recursion may also be introduced by two or more grammar rules. For example:
  $S \rightarrow Aa|b$
  $A \rightarrow Ac|Sd|\epsilon$
- Hidden left recursion due to $S \rightarrow Aa \rightarrow Sda$

Left recursion hidden due to many productions

- Left recursion may also be introduced by two or more grammar rules. For example:
  $S \rightarrow Aa|b$
  $A \rightarrow Ac|Sd|\epsilon$
- Hidden left recursion due to $S \rightarrow Aa \rightarrow Sda$
- Remove left recursion systematically.

Left recursion hidden due to many productions

- Left recursion may also be introduced by two or more grammar rules. For example:
  $S \rightarrow Aa|b$
  $A \rightarrow Ac|Sd|\epsilon$
- Hidden left recursion due to $S \rightarrow Aa \rightarrow Sda$
- Remove left recursion systematically.
  - Starting from the first rule and replacing all the occurrences of the first non terminal symbol.

Left recursion hidden due to many productions

- Left recursion may also be introduced by two or more grammar rules. For example:
  $S \rightarrow Aa|b$
  $A \rightarrow Ac|Sd|\epsilon$
- Hidden left recursion due to $S \rightarrow Aa \rightarrow Sda$
- Remove left recursion systematically.
  - Starting from the first rule and replacing all the occurrences of the first non terminal symbol.
  - Removing left recursion from the modified grammar.

Left recursion hidden due to many productions

- Left recursion may also be introduced by two or more grammar rules. For example:
  $S \rightarrow Aa|b$
  $A \rightarrow Ac|Sd|\epsilon$
- Hidden left recursion due to $S \rightarrow Aa \rightarrow Sda$
- Remove left recursion systematically.
  - Starting from the first rule and replacing all the occurrences of the first non terminal symbol.
  - Removing left recursion from the modified grammar.
- After the first step (substitute S by its rhs in the rules) the grammar becomes

Left recursion hidden due to many productions

- Left recursion may also be introduced by two or more grammar rules. For example:
  $S \rightarrow Aa|b$
  $A \rightarrow Ac|Sd|\epsilon$
- Hidden left recursion due to $S \rightarrow Aa \rightarrow Sda$
- Remove left recursion systematically.
  - Starting from the first rule and replacing all the occurrences of the first non terminal symbol.
  - Removing left recursion from the modified grammar.
- After the first step (substitute S by its rhs in the rules) the grammar becomes
  $S \rightarrow Aa|b$
  $A \rightarrow Ac|Aad|bd|\epsilon$

Left recursion hidden due to many productions

- Left recursion may also be introduced by two or more grammar rules. For example:
  $S \rightarrow Aa|b$
  $A \rightarrow Ac|Sd|\epsilon$
- Hidden left recursion due to $S \rightarrow Aa \rightarrow Sda$
- Remove left recursion systematically.
  - Starting from the first rule and replacing all the occurrences of the first non terminal symbol.
  - Removing left recursion from the modified grammar.
- After the first step (substitute S by its rhs in the rules) the grammar becomes
  $S \rightarrow Aa|b$
  $A \rightarrow Ac|Aad|bd|\epsilon$
- After the second step (removal of left recursion) the grammar becomes

Left recursion hidden due to many productions

- Left recursion may also be introduced by two or more grammar rules. For example:
  $S \to Aa|b$
  $A \to Ac|Sd|\epsilon$
- Hidden left recursion due to $S \to Aa \to Sda$
- Remove left recursion systematically.
  - Starting from the first rule and replacing all the occurrences of the first non terminal symbol.
  - Removing left recursion from the modified grammar.
- After the first step (substitute S by its rhs in the rules) the grammar becomes
  $S \to Aa|b$
  $A \to Ac|Aad|bd|\epsilon$
- After the second step (removal of left recursion) the grammar becomes
  $S \to Aa|b$
  $A \to bdA'|A'$
  $A' \to cA'|adA'|\epsilon$

Left Factoring

- In top-down parsing when it is not clear which production to choose for expansion of a symbol defer the decision till we have seen enough input.

Left Factoring

- In top-down parsing when it is not clear which production to choose for expansion of a symbol defer the decision till we have seen enough input.
- $A \rightarrow \alpha\beta_1|\alpha\beta_2$ transforms to

Left Factoring

- In top-down parsing when it is not clear which production to choose for expansion of a symbol defer the decision till we have seen enough input.
- $A \rightarrow \alpha\beta_1|\alpha\beta_2$ transforms to
- $A \rightarrow \alpha A'$
  $A' \rightarrow \beta_1|\beta_2$

three things to remove from grammar before performing top-down parsing-
1. Ambiguity
2. Left recursion
3. Left factoring