CS202 – System Software

Dr. Manish Khare

Lecture 8

NDFA to DFA Conversion

- Problem Statement
- Let $\mathbf{X = (Q_x, \sum, \delta_x, q_0, F_x)}$ be an NDFA which accepts the language L(X). We have to design an equivalent DFA $\mathbf{Y = (Q_y, \sum, \delta_y, q_0, F_y)}$ such that **L(Y) = L(X)**.

- Non-Deterministic Finite Automata,
  - For some current state and input symbol, there exists more than one next output states.
  - A string is accepted only if there exists at least one transition path starting at initial state and ending at final state.

- The following procedure converts the NDFA to its equivalent DFA −

- **Step-01:**
  - Let Q' be a new set of states of the DFA. Q' is null in the starting.
  - Let T' be a new transition table of the DFA.

- **Step-02:**
  - Add start state of the NFA to Q'.
  - Add transitions of the start state to the transition table T'.
  - If start state makes transition to multiple states for some input alphabet, then treat those multiple states as a single state in the DFA.

Step-03:

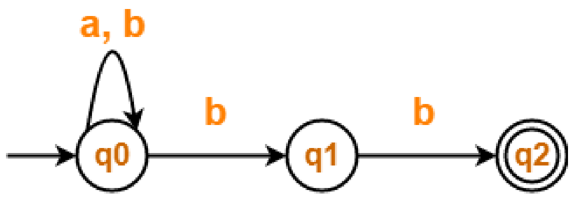
- If any new state is present in the transition table T',
- Add the new state in Q'.
- Add transitions of that state in the transition table T'.

Step-04:

- Keep repeating Step-03 until no new state is present in the transition table T'.
- Finally, the transition table T' so obtained is the complete transition table of the required DFA.

Exercise

- Convert the following Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA)-



- Transition table for the given Non-Deterministic Finite Automata (NFA) is-

State / Alphabet	**a**	**b**
→q0	q0	q0, q1
q1	–	*q2
***q2**	–	–

Step-01:

- Let Q/ be a new set of states of the Deterministic Finite Automata (DFA).
- Let T/ be a new transition table of the DFA.

Step-02:

- Add transitions of start state q0 to the transition table T/.

State / Alphabet	a	b
→q0	q0	{q0, q1}

Step-03:

- New state present in state $Q^{/}$ is {q0, q1}.
- Add transitions for set of states {q0, q1} to the transition table $T^{/}$.

State / Alphabet	**a**	**b**
→q0	q0	{q0, q1}
{q0, q1}	q0	{q0, q1, q2}

Step-04:

- New state present in state Q$^/$ is {q0, q1, q2}.
- Add transitions for set of states {q0, q1, q2} to the transition table T$^/$.

State / Alphabet	**a**	**b**
→q0	q0	{q0, q1}
{q0, q1}	q0	{q0, q1, q2}
{q0, q1, q2}	q0	{q0, q1, q2}

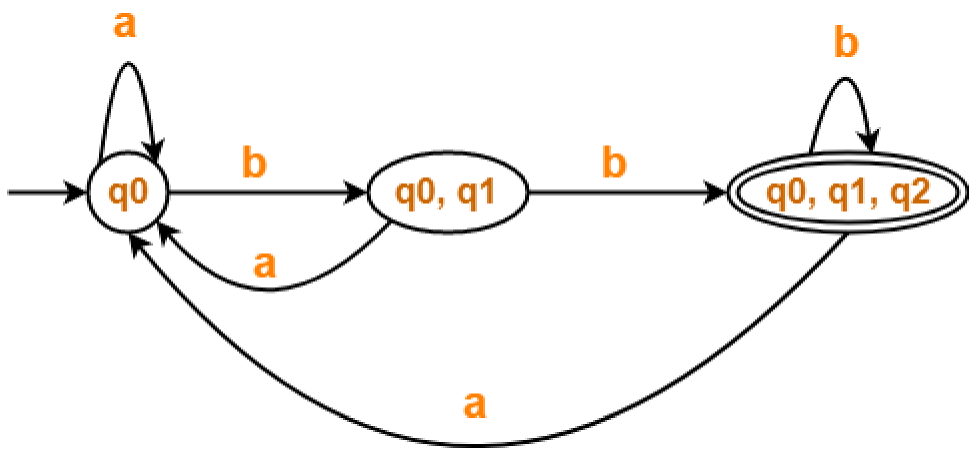
Step-05:

- Since no new states are left to be added in the transition table T’, so we stop.
- States containing q2 as its component are treated as final states of the DFA.

Finally, Transition table for Deterministic Finite Automata (DFA) is-

State / Alphabet	**a**	**b**
→q0	q0	{q0, q1}
{q0, q1}	q0	*{q0, q1, q2}
***{q0, q1, q2}**	q0	*{q0, q1, q2}

- Now, Deterministic Finite Automata (DFA) may be drawn as-



Deterministic Finite Automata (DFA)

- It is important to note the following points when converting a given NFA into a DFA-
- **Note-01:**
  - After conversion, the number of states in the resulting DFA may or may not be same as NFA.
  - The maximum number of states that may be present in the DFA are $2^{\text{Number of states in the NFA}}$.
- **Note-02:**
  - In general, the following relationship exists between the number of states in the NFA and DFA-

$$\mathbf{1 <= n <= 2^{m}}$$

Here,

•n = Number of states in the DFA

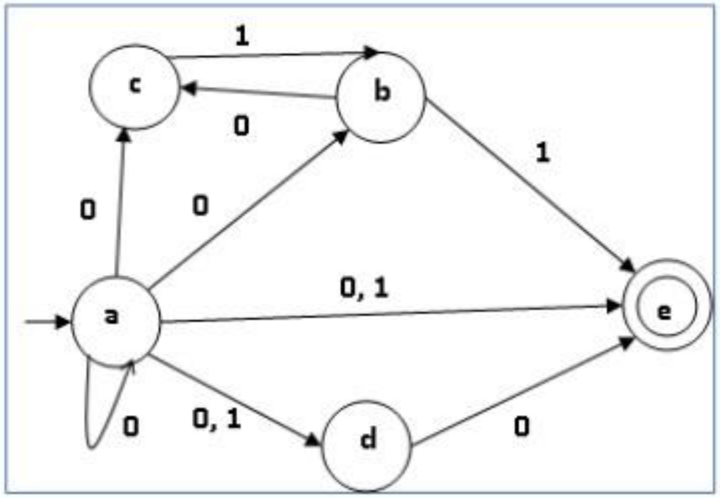
•m = Number of states in the NFA

Note-03:

- In the resulting DFA, all those states that contain the final state(s) of NFA are treated as final states.

Exercise

- Let us consider the NDFA shown in the figure below.



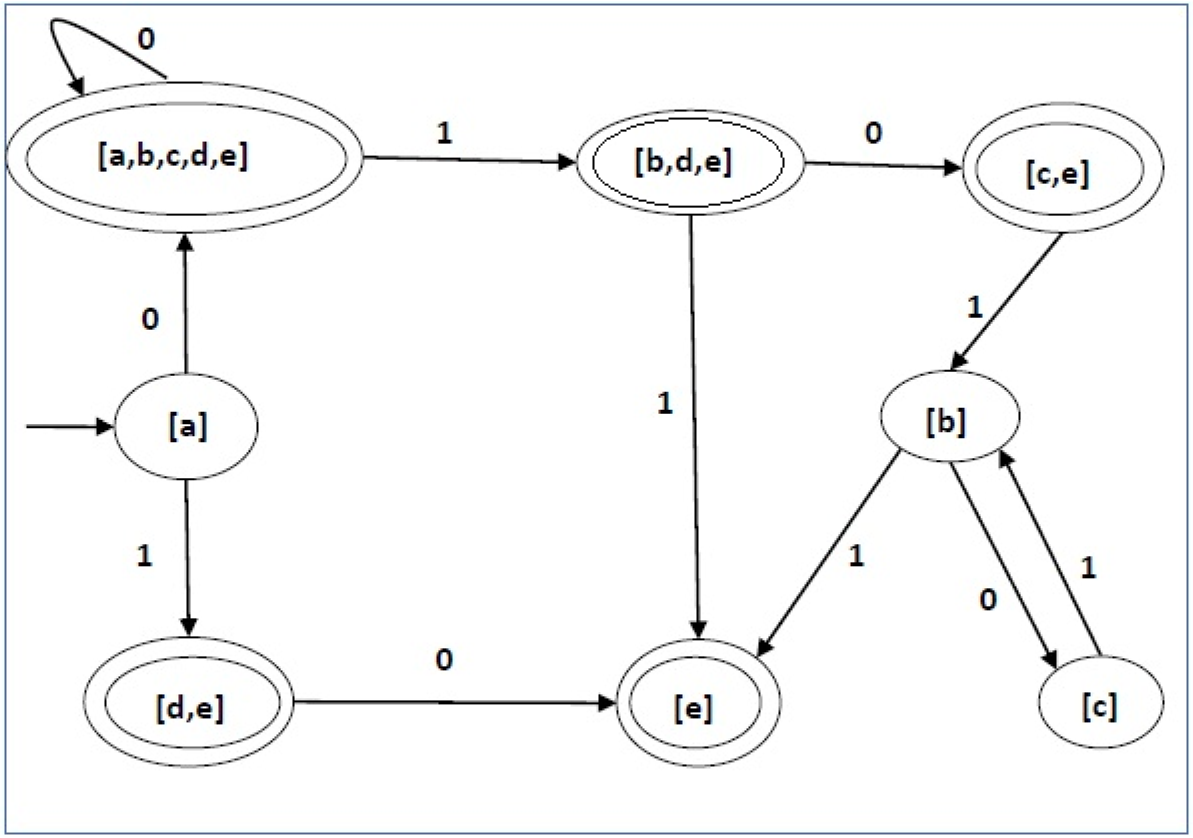
- Now we need to Convert this NDFA to DFA

q	δ(q,0)	δ(q,1)
a	{a,b,c,d,e}	{d,e}
b	{c}	{e}
c	∅	{b}
d	{e}	∅
e	∅	∅

- Using the above algorithm, we find its equivalent DFA. The state table of the DFA is shown in below.

q	δ(q,0)	δ(q,1)
[a]	[a,b,c,d,e]	[d,e]
[a,b,c,d,e]	[a,b,c,d,e]	[b,d,e]
[d,e]	[e]	∅
[b,d,e]	[c,e]	[e]
[e]	∅	∅
[c, e]	∅	[b]
[b]	[c]	[e]
[c]	∅	[b]

➢The state diagram of the DFA is as follows



Alphabet, Strings and Languages

Alphabet

- An alphabet, is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. E.g. {a,b,c}, {0,1}.
- The set {0,1} is the binary alphabet.
- ASCII is an important example of an alphabet.

Alphabet, Strings and Languages

String

- A string over some alphabet is a finite sequence of symbols drawn from that alphabet.
- In language theory, the terms sentence and word are often used as synonyms for ‘string’.
- The length of string S, usually written |S|, is the no. of occurrence of symbol in S.
- For ex. ‘**banana**’ is a string of length six.
- The empty string, denoted ε , is the string of length zero.

Alphabet, Strings and Languages

Language

- A language is any countable set of strings over some fixed alphabet.
- If ‘x’ and ‘y’ are two strings, then concatenation of ‘x’ and ‘y’ written as ‘xy’ is the string formed by appending ‘y’ to ‘y’.
- For ex. If ‘x’=‘dog’ and ‘y’=‘house’ then ‘xy’=‘dog house’ and ‘yx’=‘house dog’
- $xy \neq yx$

Alphabet, Strings and Languages

Terms for parts of strings

Terms	Definitions
Prefix of S	A string obtained by removing zero or more trailing symbols of string S, e.g. ‘ban’ is a prefix of ‘banana’
Suffix of S	A string formed by deleting zero or more of the leading symbols of S, e.g. ‘nana’ is suffix of ‘banana’
Substring of S	A string obtained by deleting a prefix and a suffix from S. e.g. ‘nan’ is a substring of ‘banana’. Every prefix and every suffix of S is a substring of S is a prefix or a suffix of S. for every string S, both S and ε are prefixes, suffixes, and substring of S.
Proper prefix, suffix or substring of S	Any nonempty string x that is, respectively a prefix, suffix or substring of S, such that S ≠ x
Subspace of x	Any string formed by deleting zero or more not necessarily consecutive positions of S. for ex. ‘baan’ is a subsequence of ‘banana’

Operation on Language

- In lexical analysis, the most important operations on languages are union, concatenation, and closure.

Union of two language

- Let two language are Land M, then union of these two language is LUM , and defined as L U M = {s|s is in L or s is in M}

Concatenation of two language

- Let two language are L and M, then concatenation of these two language is LM, and defined as LM = {st | s is in L and t is in M}

Specification of Tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
  - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

- Let L be the set of letters **{A, B, . . . , Z, a, b, . . . , *z)*** and let *D* be the set of digits **{0,1,..** .9).
- We may think of *L* and *D* in two, essentially equivalent, ways.
  - One way is that *L* and *D* are, respectively, the alphabets of uppercase and lowercase letters and of digits.
  - The second way is that *L* and *D* are languages, all of whose strings happen to be of length one.

- Here are some other languages that can be constructed from languages *L* and *D*, using the operators described earlier.

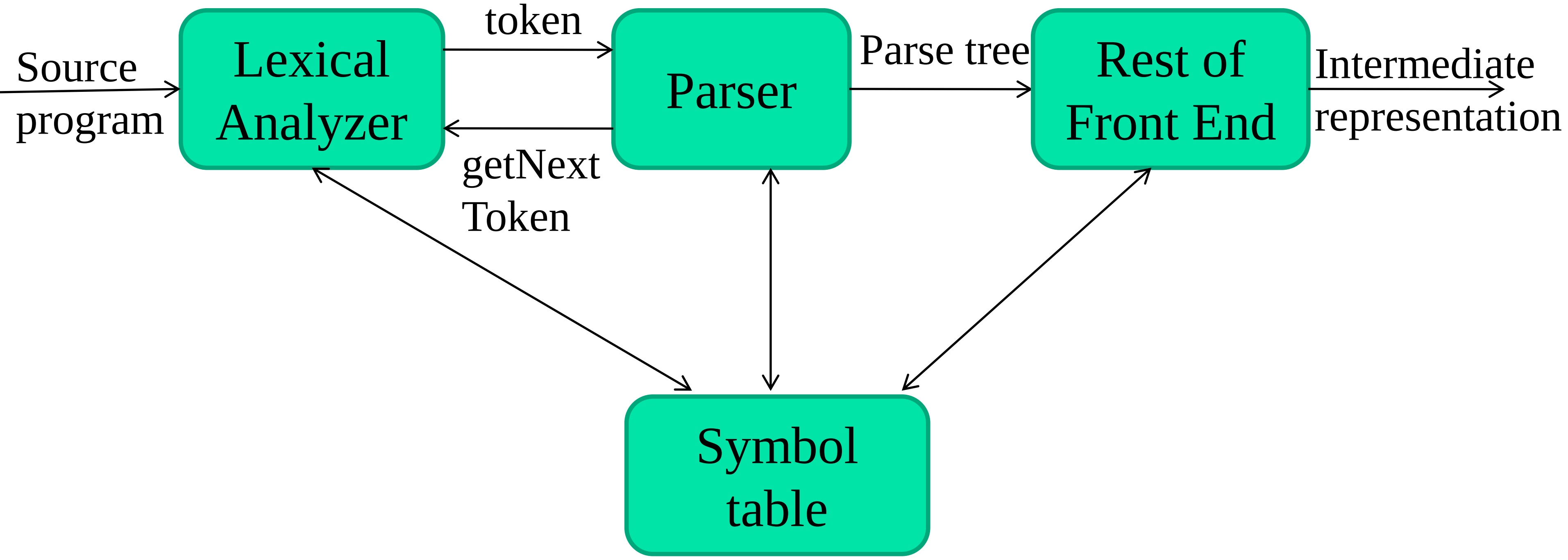
- 1. L *U D* is the set of letters and digits - strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
- 2. *LD* is the set of 520 strings of length two, each consisting of one letter followed by one digit.
- 3. *L4* is the set of all 4-letter strings.
- *4. L** is the set of ail strings of letters, including e, the empty string.
- 5. *L(L U D)** is the set of all strings of letters and digits beginning with a
- letter.
- 6. *D+* is the set of all strings of one or more digits.

Syntax Analyzer

Syntax Analyzer

- Role of parser
- Types of errors
- Parse tree
- Top down parsing
- Bottom up parsing
- Parser generators

Role of Parser



Role of Parser

- Parser plays an important role in the compiler design.
  - It obtains a string of tokens from the lexical analyzer.
  - It groups the tokens in order to identify larger structures in the program.
  - It should report any syntax error in the program.
  - It should also recover from the errors so that it can continue to process the rest of the input.

Type of errors

- If a compiler had to process only correct program, its design and implementation would be greatly simplified, but programmers frequently write incorrect programs and good compiler should assist the programmer to identifying and locating error.

- Program can contains following types of error
  - **Lexical error** – such as misspelling an identifier, keyword, or operator.
  - **Syntactic error** – such as arithmetic expression with unbalanced parenthesis.
  - **Semantic error** – such as operator, applied to an incompatible operand.
  - **Logical error** – such as an identify recursive cells.

Error handler

- The error handler in a process has simple to state goals.
  - It should report the presence of errors clearly and accurately.
  - It should recover from each error quickly enough to be able to detect subsequent errors.
  - It should not significantly slow down processing of correct programs

Error recovery strategies

- There are many strategies, that a parser can employ to recover from a syntactic error.
- Although no one strategy has proven itself to be universally acceptable
- Three popular strategy are given below
  - Panic mode
  - Global correction
  - Phrase level

Panic mode error recovery strategy

- This is the simplest method to implement and can be used by most parsing methods on discovering an error.
- The parser discards input symbol one at a time until one of a designated set of synchronizing token is found.

Global correction error recovery strategy

- The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free.
- When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y.
- This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Phrase level error recovery strategy

- On discovering an error, a parser may perform local correction on the remaining part, that is it may replace a prefix of the remaining input by some string that allows the parser to continue.
- A typical local correction would be replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.

Context Free Grammar

- In general grammar involves four quantities: terminals, non-terminals, a start symbol, and productions.
  - The basic symbol of which strings in the language are composed are called terminals. The ‘token’ is a synonym for ‘terminal’.
  - ‘Non-terminal’ are special symbols that denote set of strings.
  - One non-terminal is selected as the start symbol
  - The productions define the ways in which the syntactic categories may be built up from one another and from the terminals. Each production consists of a non-terminal

Context Free Grammar

- *Context-free syntax* is specified with a *context-free grammar*.

- Formally a CFG G = (V_t,V_n,S,P), where:
  - V_t is the set of ***terminal*** symbols in the grammar (i.e.,the set of tokens returned by the scanner)
  - V_n, the ***non-terminals***, are variables that denote sets of (sub)strings occurring in the language. These impose a structure on the grammar.
  - S is the ***goal symbol***, a distinguished non-terminal in V_n denoting the entire set of strings in L(G).
  - P is a finite set of ***productions*** specifying how terminals and non-terminals can be combined to form strings in the language.
    Each production must have a single non-terminal on its left hand side.

Syntax Definition: Grammars

- A grammar is a 4-tuple G = (N, T, P, S) where
  - T is a finite set of tokens (terminal symbols)
  - N is a finite set of nonterminals
  - P is a finite set of productions of the form $\alpha \rightarrow \beta$

    where $\alpha \in (N \cup T)^* N (N \cup T)^*$ and $\beta \in (N \cup T)^*$
  - $S \in N$ is a designated start symbol

- A* is the set of finite sequences of elements of A. If A = {a,b}, A* = {ε, a, b, aa, ab, ba, bb, aaa, …}

- $AB = \{ab \mid a \in A, b \in B\}$

Notational Conventions Used

- Terminals
  - $a,b,c,\ldots \in T$
  - specific terminals: 0, 1, id
- Nonterminals
  - $A,B,C,\ldots \in N$
  - specific nonterminals: expr, term, stmt
- Grammar symbols
  - $X,Y,Z \in (N \cup T)$
- Strings of terminals
  - $u,v,w,x,y,z \in T^*$
- Strings of grammar symbols
  - $\alpha,\beta,\gamma \in (N \cup T)^*$

Derivations

- The production rules are used to derive certain strings. The generation of language using specific rules is called derivation.

Derivations

- A one-step derivation is defined by
  - $\gamma\ \alpha\ \delta \Rightarrow \gamma\ \beta\ \delta$

  where $\alpha \rightarrow \beta$ is a production in the grammar
- In addition, we define
  - $\Rightarrow$ is leftmost $\Rightarrow_{lm}$ if γ does not contain a nonterminal
  - $\Rightarrow$ is rightmost $\Rightarrow_{rm}$ if δ does not contain a nonterminal
  - Transitive closure $\Rightarrow^*$ (zero or more steps)
  - Positive closure $\Rightarrow^+$ (one or more steps)
- α is a sentential form if $S \Rightarrow^* \alpha$
- The language generated by G is defined by $L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$

Derivations (Example)

Grammar $G = (\{E\}, \{+,*,(,),-,\mathbf{id}\}, P, E)$ with productions

$$P = \begin{array}{l} E \rightarrow E + E \\ E \rightarrow E * E \\ E \rightarrow (E) \\ E \rightarrow - E \\ E \rightarrow \mathbf{id} \end{array}$$

Example derivations:

$$E \Rightarrow - E \Rightarrow - \mathbf{id}$$

$$E \Rightarrow_{rm} E + E \Rightarrow_{rm} E + \mathbf{id} \Rightarrow_{rm} \mathbf{id} + \mathbf{id}$$

$$E \Rightarrow^{*} E$$

$$E \Rightarrow^{*} \mathbf{id} + \mathbf{id}$$

$$E \Rightarrow^{+} \mathbf{id} * \mathbf{id} + \mathbf{id}$$

Another Grammar for expressions

$$G = <\{list, digit\}, \{\mathbf{+}, \mathbf{-}, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{9}\}, P, list>$$

Productions $P =$

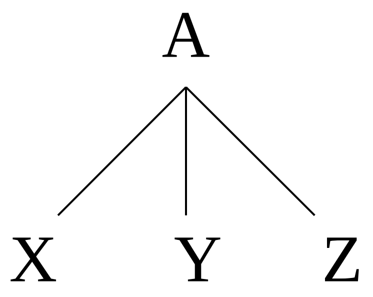
$$list \rightarrow list \; \mathbf{+} \; digit$$
$$list \rightarrow list \; \mathbf{-} \; digit$$
$$list \rightarrow digit$$
$$digit \rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

A *leftmost derivation*:

$$\begin{aligned}
&\underline{list} \\
&\Rightarrow_{lm} \underline{list} \; \mathbf{+} \; digit \\
&\Rightarrow_{lm} \underline{list} \; \mathbf{-} \; digit \; \mathbf{+} \; digit \\
&\Rightarrow_{lm} \underline{digit} \; \mathbf{-} \; digit \; \mathbf{+} \; digit \\
&\Rightarrow_{lm} \mathbf{9} \; \mathbf{-} \; \underline{digit} \; \mathbf{+} \; digit \\
&\Rightarrow_{lm} \mathbf{9} \; \mathbf{-} \; \mathbf{5} \; \mathbf{+} \; \underline{digit} \\
&\Rightarrow_{lm} \mathbf{9} \; \mathbf{-} \; \mathbf{5} \; \mathbf{+} \; \mathbf{2}
\end{aligned}$$

Parse tree

- A parse tree may be viewed as graphical representation for a derivation that filters out the choice regarding replacement order. If non-terminal A has the production A → XYZ, then parse tree may have an interior node labeled A with three children labeled X, Y, and Z from left to right

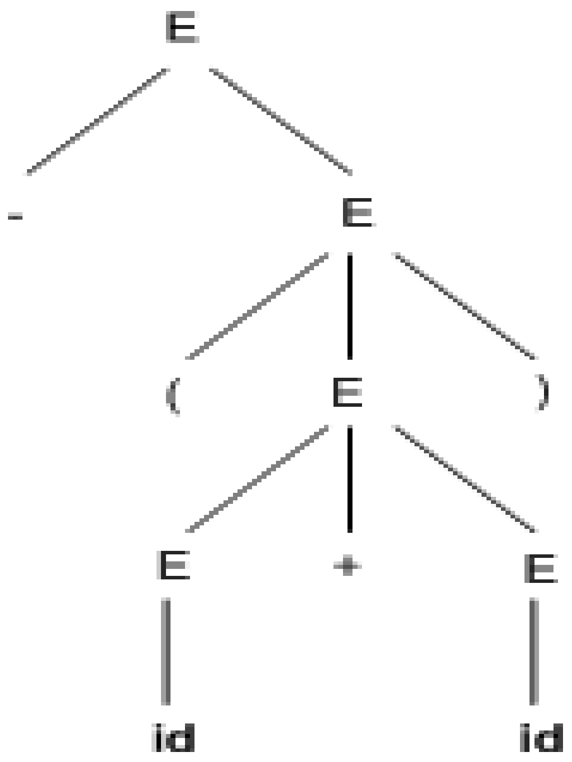


Parse tree

- A parse tree for a context free grammar G has the following characteristics
  - The root is labeled by the start symbol.
  - Each leaf is labeled by the token or by ε.
  - Each interior node is labeled by a non-terminal
  - If A is the non-terminal labeling some interior node, and X1,.. Xn, are the labels of the children of that node from left to right, then A → X1...Xn is a production. Here these X1,.. Xn stands for a symbol that is either a terminal or a non-terminal. As a special case, if A → ε, then a node labeled, A may have a single child labeled ε.

Parse tree

- For ex. Parse tree for -(**id**+**id**)   if E $\longrightarrow$ id
  - E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)



Parse tree

- Parse tree ignores variations in the order in which symbols in sentential forms are replaced.
- The variation in the order in which production are applied can also be eliminated by consider only left most (or right most) derivations.
- Every parse tree has associated with its unique left most and a unique right most derivation.

1. *Leftmost derivation:* replace the leftmost non-terminal at each step
2. *Rightmost derivation:* replace the rightmost non-terminal at each step