Course Outcomes

CO 1: Acquire fundamental understanding of the core concepts in automata theory and Theory of Computation

CO 2: Design and develop lexical analyzers, parsers and code generators

CO 3: Design Grammars and Automata (recognizers) for different language classes and become knowledgeable about restricted models of Computation (Regular, Context Free) and their relative powers.

CO 4: Acquire fundamental understanding of the structure of a Compiler and Apply concepts automata theory and Theory of Computation to design Compilers

CO 5: Design computations models for problems in Automata theory and adaptation of such model in the field of compilers

Institution Vision

To produce top-quality engineers who are groomed for attaining excellence in their profession and competitive enough to help in the growth of nation and global society.

Institution Mission

M1: To offer affordable high-quality graduate program in engineering with value education and make the students socially responsible.

M2: To support and enhance the institutional environment to attain research excellence in both faculty and students and to inspire them to push the boundaries of knowledge base.

M3: To identify the common areas of interest amongst the individuals for the effective industry-institute partnership in a sustainable way by systematically working together.

M4: To promote the entrepreneurial attitude and inculcate innovative ideas among the engineering professionals.

Department Vision

To be a center of excellence in Information Science & Engineering education, research and training to meet the growing needs of the industry and society.

Department Mission

M1: To impart theoretical and practical knowledge through the concepts and technologies in Information Science and Engineering

M2: To foster research, collaboration and higher education with premier institutions and industries.

M3: Promote innovation and entrepreneurship to fulfill the needs of the society and industry

Program Educational Objectives

PEO1: Analyse, design and implement solutions to the real-world problems in the field of Information Science and Engineering with multidisciplinary setup

PEO2: Keep abreast with the technology, innovation and pursue higher education with high standards of social and professional ethics

PEO3: Develop professional and entrepreneurship skills to work effectively as an individual and in a team to meet the ever-changing goals of the organization

Program Outcomes

PO1: **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.

PO2: **Problem Analysis:** Identify, formulate, review research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural science and engineering sciences.

PO3: **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal and environmental considerations.

PO4: **Conduct investigations of complex problems:** Use research based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: **Modern tool usage:** Create, select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations

PO6: **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice

PO7: **Environment sustainability:** Understand the impact of the professional engineering solutions in the societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions

PO11: **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to ones own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: **Lifelong learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broader context of technological change.

Program Specific Outcomes

PSO1: Design, implement and maintain the information systems that fulfill the current needs of the industry and society

PSO2: Apply computational theory, storage and networking concepts to solve the day to day problems of the world

Context free grammar

A *context free grammar* (or CFG) to be a grammar in which each rule must:

- have a left-hand side that is a single nonterminal, and
- have a right-hand side.

Define an *A* rule, for any nonterminal symbol *A,* to be a rule whose left·hand side is *A.*

Next define a control algorithm of the sort. A derivation will halt whenever no rule's left-hand side matches against *working-string.* At every step, any rule that matches may be chosen.

Context-free grammar rules may have any (possibly empty) sequence of symbols on the right-band side. Because the rule format is more flexible than it is for regular grammars. the rules are more powerful.

All of the following are allowable context-free grammar rules (assuming appropriate alphabets):

S→aSb

S→€

T→T

S→aSbbTT

The following are not allowable context-free grammar rules:

ST→aSb

a→ aSb

€→a

The name for these grammars, "context-free," makes sense because, using these rules, the decision to replace a nonterminal by some other sequence is made without looking at the context in which the non terminal occurs.

Definition:

Context Free grammar or CGF, G is represented by four components that is G=(V,T,P,S), where V is the set of variables, T the terminals, P the set of productions and S the start symbol.

Example:

The grammar Gpal for palindromes is represented by Gpal = ({P},{0,1}, A, P) where A represents the set of five productions

1. $P \rightarrow \varepsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

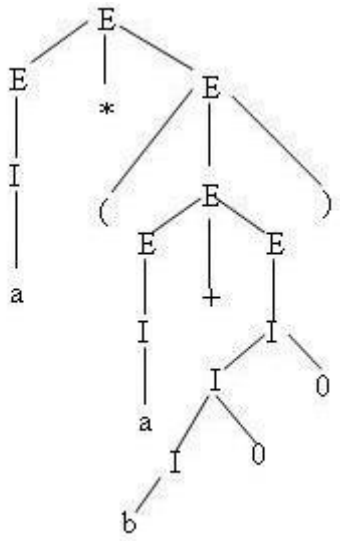
There are four important components in a grammatical description of a language.

1. There is a finite set of symbols that form the strings of the language being defined. This set was {0,1} in the palindrome example we just saw. We call this alphabet the terminals, or terminal symbols.
2. There is a finite set of variables, also called sometimes nonterminals or syntactic categories. Each variable represents a language i.e. a set of strings. In our example above, there was only one variable, P, which we used to represent the class of palindromes over alphabet {0,1}.
3. One of the variables represents the language being defined it is called the start symbol. Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol. In our example, P, the only variable, is the start symbol.
4. There is a finite set of productions or rules that represent the recursive definition of a language. Each production consists of:
   a. A variable that is being {partially} defined by the production. This variable is often called the head of the production.
   b. The production symbol →.
   c. A string of zero or more terminals and variables. This string, called the body of the production, represents one way to form strings in the language of the variable of the head_ In so doing, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in the language of that variable

Derivation using Grammar

Consider a context-free grammar for simple expressions

1. E→ I
2. E→ E + E
3. E→ E * E
4. E→ (E)
5. I→ a
6. I→ b
7. I→ Ia
8. I→ Ib
9. I→ I0
10. I→ I1



Parse trees

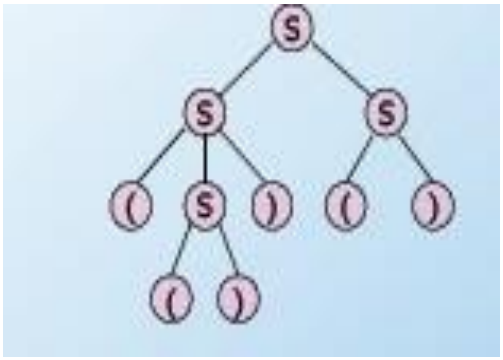
Parse trees are trees labeled by symbols of a particular CFG. Leaves: labeled by a terminal or ε. Interior nodes: labeled by a variable. Children are labeled by the right side of a production for the parent.

Root: must be labeled by the start symbol

Example: Parse Tree

S -> SS | (S) | ()

Input: (())()



Leftmost Derivation

The inference that a * (a+b00) is in the language of variable E can be reflected in a derivation of that string, starting with the string E.

E→I

E→E+E

E→E*E

E→(E)

I→a

I→b

I→Ia

I→Ib

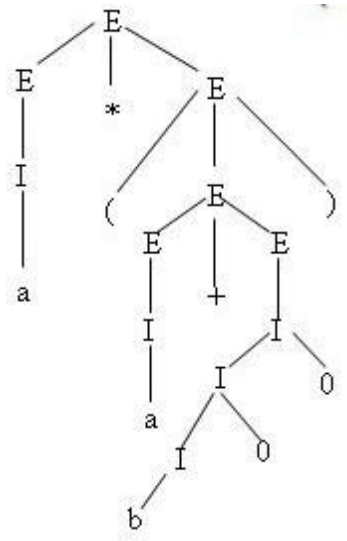
I→I0

I→I1

Here is one such derivation:

E →E * E → I * E → a * E →a * (E) → a * (E + E) → a * (I + E) → a * (a + E) →a * (a + I) → a * (a + I0) → a * (a + I00) → a * (a + b00)

Leftmost Derivation - Tree

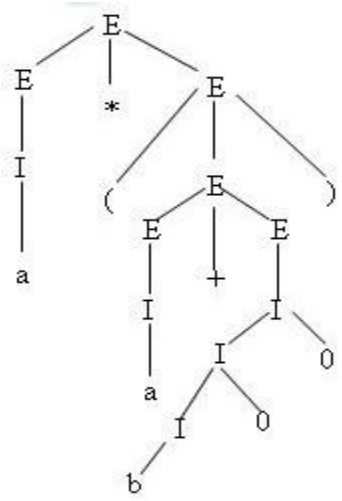


Rightmost Derivations

The derivation of Example 1 was actually a leftmost derivation. Thus, we can describe the same derivation by:

E→ E * E → E *(E) → E * (E + E) →E * (E + I) → E * (E +I0) → E * (E + I00) → E * (E + b00) →E * (I + b00) → E * (a +b00) → I * (a + b00) → a * (a + b00) We can also summarize the leftmost derivation by saying E → a * (a + b00), or express several steps of the derivation by expressions such as E * E → a * (E).

Rightmost Derivation - Tree



There is a rightmost derivation that uses the same replacements for each variable, although it makes the replacements in different order. This rightmost derivation is:

E → E * E → E * (E) → E * (E + E) →E * (E + I) → E * (E + I0) → E * (E + I00) → E * (E + b00) →E * (I + b00) → E * (a + b00) → I * (a + b00) → a * (a + b00) This derivation allows us to conclude E → a * (a + b00)

Consider the Grammar for string(a+b)*c

E→E + T | T

T→ T * F | F

F→ (E) | a | b | c

Leftmost Derivation

E→T→T*F→F*F→(E)*F→(E+T)*F→(T+T)*F→(F+T)*F →(a+T)*F →(a+F)*F →(a+b)*F→(a+b)*c

Rightmost derivation

E→T→T*F→T*c→F*c→(E)*c→(E+T)*c→(E+F)*c→(E+b)*c→(T+b)*c→(F+b)*c→(a+b)*c

Example 2:

Consider the Grammar for string (a,a)

S->(L)|a

L->L,S|S

Leftmost derivation

S→ (L)→(L,S)→(S,S)→(a,S)→(a,a)

Rightmost Derivation

S→(L)→(L,S)→(L,a)→(S,a)→(a,a)

The Language of a Grammar

If G(V,T,P,S) is a CFG, the language of G, denoted by L(G), is the set of terminal strings that have derivations from the start symbol.

$L(G) = \{w \text{ in } T \mid S \rightarrow w\}$

Sentential Forms

Derivations from the start symbol produce strings that have a special role called "sentential forms". That is if G = (V, T, P, S) is a CFG, then any string in $(V \cup T)^*$ such that $S \rightarrow \alpha$ is a sentential form. If $S \rightarrow \alpha$, then is a left – sentential form, and if $S \rightarrow \alpha$, then is a right – sentential form. Note that the language L(G) is those sentential forms that are in T^*; that is they consist solely of terminals.

For example, E * (I + E) is a sentential form, since there is a derivation

$E \rightarrow E * E \rightarrow E * (E) \rightarrow E * (E + E) \rightarrow E * (I + E)$

However, this derivation is neither leftmost nor rightmost, since at the last step, the middle E is replaced.

As an example of a left – sentential form, consider a * E, with the leftmost derivation.

$E \rightarrow E * E \rightarrow I * E \rightarrow a * E$

Additionally, the derivation

$E \rightarrow E * E \rightarrow E * (E) \rightarrow E * (E + E)$ Shows that E * (E + E) is a right – sentential form.

Ambiguity in Grammars and Languages

A context – free grammar G is said to be ambiguous if there exists some w ∈ L(G) which has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of more left most or rightmost derivations.

Ex:-

Consider the grammar G=(V,T,E,P) with V={E,I},

E→I,

E→E+E,

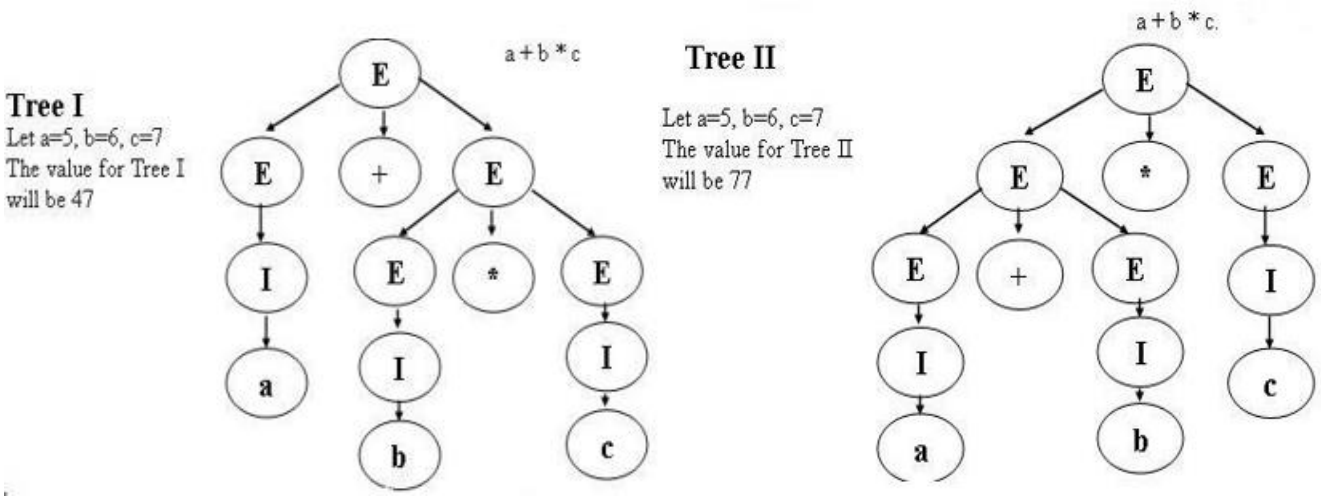
E→E*E

E→(E),

I→a|b|c

Consider two derivation trees for a + b * c.

T={a,b,c,+,*,(,)}, and productions



Removing Ambiguity from Grammars

There are two causes of ambiguity in the grammar of Fig:

1. The precedence of operators is not respected. While tree 1 properly groups the * before the + operator. Tree 2 is also a valid parse tree and groups the + ahead of the *. We need to force only the structure of tree 1 to be legal in an unambiguous grammar.

2. A sequence of identical operators can group either from the left or from the right. For example, if the *s in Fig were replaced by +s, we would see two different parse trees for the string E→E+E Since addition and multiplication are associative, it doesn't matter whether we group from the left or the

right, but to eliminate ambiguity, we must pick one. The conventional approach is to insist on grouping from the left, so the structure of Fig.

After applying precedence and associativity, the resultant grammar will be:

E→E+T

E→T

T→T*F

T→F

F→(E)

F→I

I→a|b|c

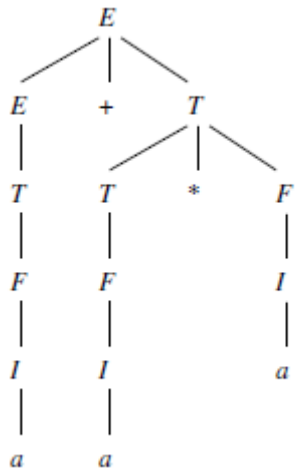


Figure: The sole parse tree for a + a * a

Inherent Ambiguity

A CFL L is said to be inherently ambiguous if all its grammars are ambiguous. If even one grammar for L is unambiguous, then L is an unambiguous language. We shall not prove that there are inherently ambiguous languages

Example:

The language L in question is:

$L=\{a^n b^n c^m d^m \mid n\geq 1, m\geq 1\} \cup L=\{a^n b^m c^m d^n \mid n\geq 1, m\geq 1\}$

Consider the Grammar for above language

S→AB | C

A→ aAb | ab

B→ cBd | cd

C→ aCd | aDd

D→ bDc | bc

This grammar is ambiguous. For example, the string aabbccdd has the two leftmost derivations.

$$1.\ S \underset{lm}{\Rightarrow} AB \underset{lm}{\Rightarrow} aAbB \underset{lm}{\Rightarrow} aabbB \underset{lm}{\Rightarrow} aabbcBd \underset{lm}{\Rightarrow} aabbccdd$$

$$2.\ S \underset{lm}{\Rightarrow} C \underset{lm}{\Rightarrow} aCd \underset{lm}{\Rightarrow} aaDdd \underset{lm}{\Rightarrow} aabDcdd \underset{lm}{\Rightarrow} aabbccdd$$

Parse tree for string aabbccdd

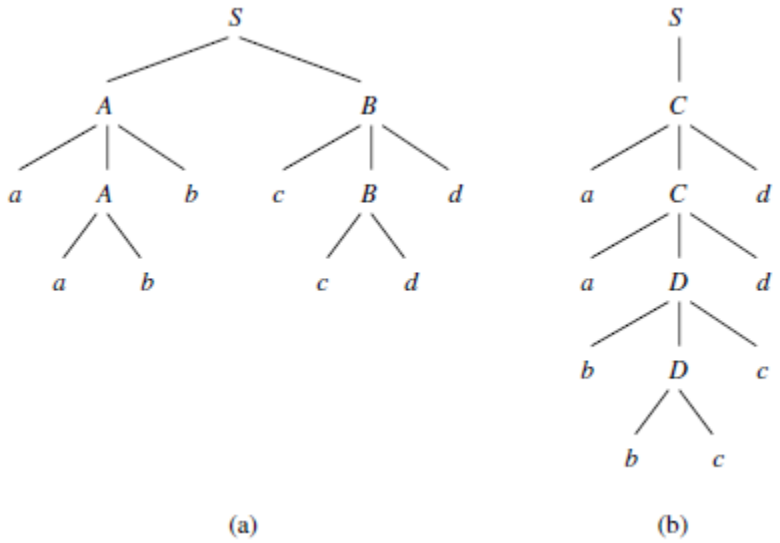


Figure: Two parse trees for aabbccdd

Elimination of Left Recursion

A Grammar G (V, T, P, S) is left recursive if it has a production in the form.

A → A α |β.

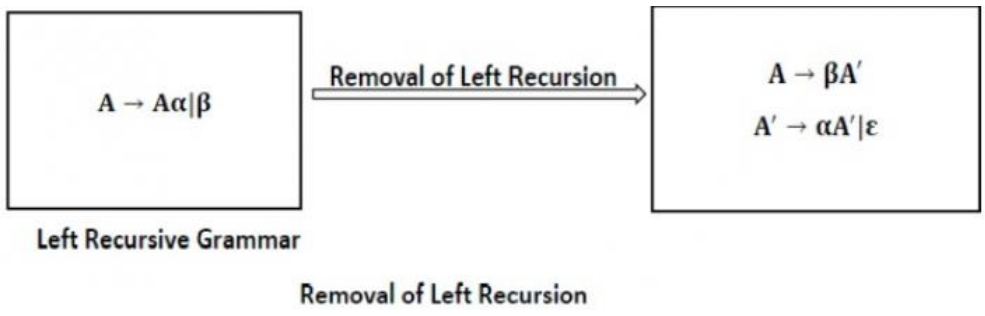
The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with

A → βA′

A → αA′|ϵ

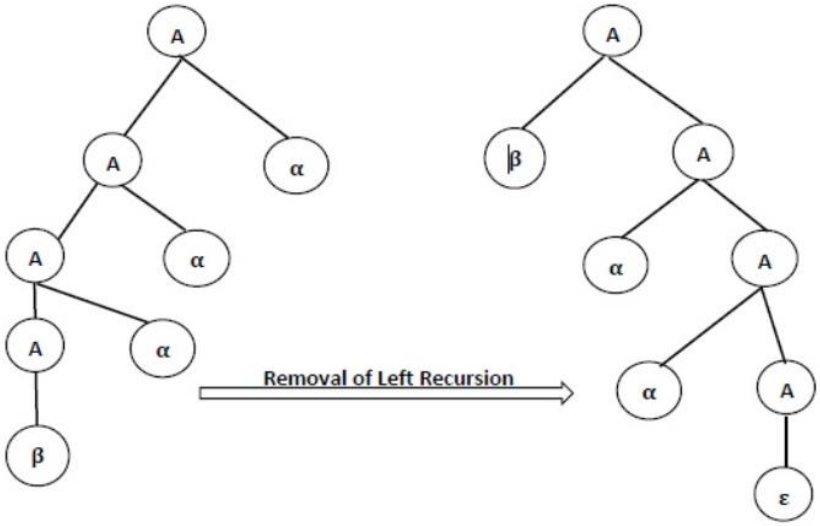
Elimination of Left Recursion

Left Recursion can be eliminated by introducing new non-terminal A such that.



This type of recursion is also called **Immediate Left Recursion**.

In Left Recursive Grammar, expansion of A will generate Aα, Aαα, Aααα at each step, causing it to enter into an infinite loop



The general form for left recursion is

$$A \rightarrow A\alpha_1|A\alpha_2| \ldots . |A\alpha_m|\beta_1|\beta_2| \ldots . . \beta_n$$

can be replaced by

$$A \rightarrow \beta_1 A'|\beta_2 A'| \ldots . . | \ldots . . |\beta_n A'$$

$$A \rightarrow \alpha_1 A'|\alpha_2 A'| \ldots . . |\alpha_m A'|\varepsilon$$

Example1 − Consider the Left Recursion from the Grammar.

$E \rightarrow E + T|T$

$T \rightarrow T * F|F$

$F \rightarrow (E)|id$

Eliminate immediate left recursion from the Grammar.

Solution

Comparing E → E + T|T with A → A α |β

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

$$A \rightarrow A\ \alpha \mid \beta$$

∴ A = E, α = +T, β = T

∴ A → A α |β is changed to A → βA′and A′ → α A′|ε

∴ A → βA′ means E → TE′

A′ → α A′|ε means E′ → +TE′|ε

Comparing T → T ∗ F|F with A → Aα|β

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

$$A \rightarrow A\ \alpha \mid \beta$$

∴ A = T, α =∗ F, β = F

∴ A → β A′ means T → FT′

A → α A′|ε means T′ →* FT′|ε

Production F → (E)|id does not have any left recursion

∴ Combining productions 1, 2, 3, 4, 5, we get

E → TE′

E′ → +TE′| ε

T → FT′

T →* FT′|ε

F → (E)| id

Example2 − Eliminate the left recursion for the following Grammar.

S → a|^|(T)

T → T, S|S

Solution

We have immediate left recursion in T-productions.

Comparing T → T, S|S With A → A α | β where A = T, α =, S and β = S

Therefore

∴ Complete Grammar will be

S→ a|^(T)

T→ ST′

T′ →,ST′| ε

Example3 − Eliminate the left recursion from the grammar

E → E + T|T

T → T * F|F

F → (E)|id

Solution

The production after removing the left recursion will be

E → TE′

E′ → +TE′| ∈

T → FT′

T′ →∗ FT′| ∈

F → (E)|id

Example4 − Remove the left recursion from the grammar

E → E(T)|T

T → T(F)|F

F → id

Solution

Eliminating immediate left-recursion among all Aα productions, we obtain

E → TE′

E → (T)E′|ε

T → FT′

T′ → (F)T′|ε

F → id

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing.
- When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

For example, if we have the two productions

$$\begin{array}{lcl} \textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\ & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \end{array}$$

on seeing the input **if**, we cannot immediately tell which production to choose to expand *stmt*. In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$. However, we may defer the decision by expanding A to $\alpha A'$. Then, after seeing the input derived from α, we expand A' to β_1 or to β_2. That is, left-factored, the original productions become

$$\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \end{array}$$

Algorithm : Left factoring a grammar.

INPUT: Grammar G.

OUTPUT: An equivalent left-factored grammar.

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$A \to \alpha A' \mid \gamma$$
$$A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. □

Example : The following grammar abstracts the "dangling-else" problem:

$$S \to i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a$$
$$E \to b$$

Here, i, t, and e stand for **if**, **then**, and **else**; E and S stand for "conditional expression" and "statement." Left-factored, this grammar becomes:

$$S \to i\ E\ t\ S\ S' \mid a$$
$$S' \to e\ S \mid \epsilon$$
$$E \to b$$

Thus, we may expand S to $iEtSS'$ on input i, and wait until $iEtS$ has been seen to decide whether to expand S' to eS or to ϵ. Of course, these grammars are both ambiguous, and on input e, it will not be clear which alternative for S' should be chosen.

Example:

Do left factoring in the following grammar-

A → aAB | aBc | aAc

Solution-

Step-01:

A → aA'

A' → AB / Bc / Ac

Again, this is a grammar with common prefixes.

Step-02:

$$A \rightarrow aA'$$

$$A' \rightarrow AD / Bc$$

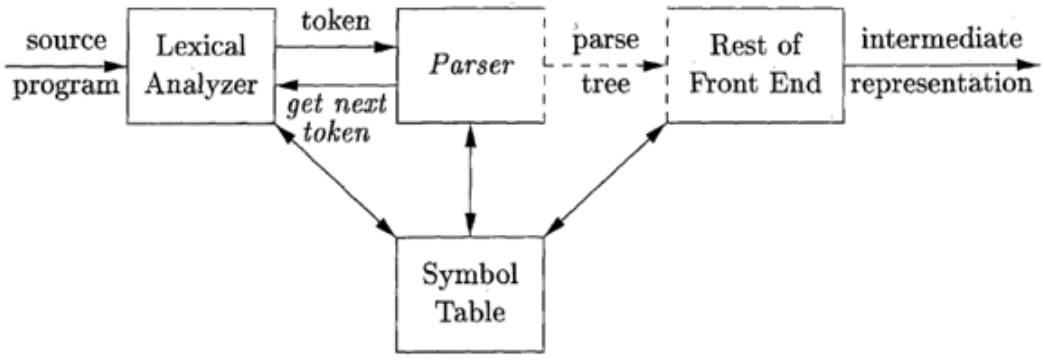
$$D \rightarrow B / c$$

This is a left factored grammar.

Syntax Analysis

Syntax analysis. The role of parser.

- Syntax Analysis or Parsing is the second phase of compiler, i.e. after lexical analysis.
- It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not.
- It does so by building a data structure, called a Parse tree or Syntax tree.
- The parse tree is constructed by using the pre-defined Grammar of the language and the input string.
- If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. If not, error is reported by syntax analyzer



Position of parser in compiler model

Types of parsers for grammar.

- Universal parsers: can parse any grammar;

  ex: Cocke-younger-Kasami and Early algorithm
- Top-down: build parse tree from root to the leaves, called as LL parsers.
- Bottom-up: build parse tree from leaves to the root, called as LR parsers.

  LL: Input for the parser is scanned from left to right; one symbol at a time and it is LMD.

  LR: Input for the parser is scanned from left to right; one symbol at a time and it is RMD.

Syntax Error Handling. Different error recovery strategies.

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the

input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- **Lexical** : Name of some identifier typed incorrectly
- **Syntactical** : missing semicolon or unbalanced parenthesis
- **Semantic** : Incompatible value assignment
- **Logical** : Code not reachable, infinite loop

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Phrase-Level Recovery

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example: inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

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.

Context Free Grammars

Consider a conditional statement If S1 and S2 are statements and E is an expression, then

Stmt →if E then S1 else S2.

We know that, regular expression can specify the lexical structure of tokens. Using some syntactic variable, stmt, we can specify grammar production

Stmt →if expr then stmt else stmt.

A context-free grammar consists of terminals, non-terminals, start symbol and set of productions.

1. *Terminals* are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer. The terminals are the keywords if and else and the symbols "(" and “)”.

2. *Non-terminals* are syntactic variables that denote sets of strings. *stmt* and *expr* are non-terminals. The sets of strings denoted by non-terminals help define the language generated by the grammar. Non-terminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

3. In a grammar, one nonterminal is distinguished as the *start symbol*, and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.

4. The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each *production* consists of:

(a) A nonterminal called the *head* or *left side* of the production; this production defines some of the strings denoted by the head.

(b) The symbol →.

(c) A *body* or *right side* consisting of zero or more terminals and non -terminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

Consider the grammar for simple arithmetic expressions as follows:

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

In this grammar, E, T and F are non-terminals; +,-, (,), id are terminals; E is start symbol.

Writing a grammar

Parse tree and Derivation

Example: Consider the following context free grammar E→E+E|E*E|-E|(E)|id and the input string – (id + id).

(i) Give LMD and RMD

(ii) Parse tree

(iii) Is the grammar ambiguous? why

Derivation: A **derivation** is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input: Deciding the non-terminal which is to be replaced. Deciding the production rule, by which, the non-terminal will be replaced.

1. In *leftmost* derivations, the leftmost nonterminal in each sentential is always chosen. If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal in α is replaced, we write $\alpha \underset{lm}{\Rightarrow} \beta$.

2. In *rightmost* derivations, the rightmost nonterminal is always chosen; we write $\alpha \underset{rm}{\Rightarrow} \beta$ in this case.

Left most derivation (LMD)

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$

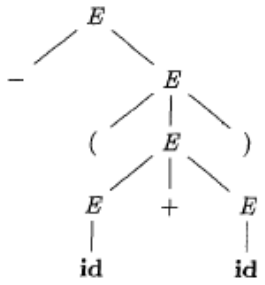
Right most derivation (RMD)

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Parse tree

A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non-terminals. Each interior node of a parse tree represents the application of a production. The interior node is labeled with the non- terminal A in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.

For example: the parse tree for – **(id + id).**



Parse tree for $-(\mathbf{id} + \mathbf{id})$

Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be *ambiguous.*
- Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

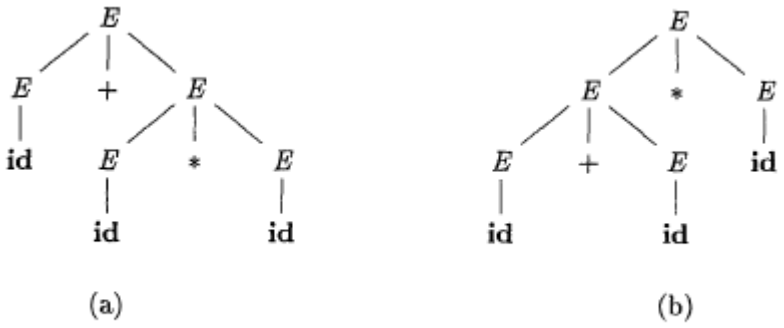
The arithmetic expression grammar

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

Permits two distinct leftmost derivations for the sentence: **id + id * id.**

$$\begin{aligned} E &\Rightarrow E+E \\ &\Rightarrow \mathbf{id}+E \\ &\Rightarrow \mathbf{id}+E*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id} \end{aligned} \qquad \begin{aligned} E &\Rightarrow E*E \\ &\Rightarrow E+E*E \\ &\Rightarrow \mathbf{id}+E*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id} \end{aligned}$$

The corresponding parse trees are



Two parse trees for **id+id*id**

Therefore the above grammar is ambiguous.

Writing a Grammar

Grammars are capable of describing most, but not all, of the syntax of programming languages. For instance, the requirement that identifiers be declared before they are used cannot be described by a context-free grammar. Therefore, the sequences of tokens accepted by a parser form a superset of the programming language; subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

Lexical Versus Syntactic Analysis

Everything that can be described by a regular expression can also be described by a grammar. We may therefore reasonably ask: ***"Why use regular expressions to define the lexical syntax of a language?"***

There are several reasons.

1. Separating the syntactic structure of a language into lexical and non-lexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

There are no firm guidelines as to what to put into the lexical rules, as opposed to the syntactic rules. Regular expressions are most useful for describing the structure of constructs such as identifiers, constants, keywords, and white space.

Grammars, on the other hand, are most useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on. These nested structures cannot be described by regular expressions.

Example: Define Ambiguity. Is the following grammar ambiguous? If yes remove the ambiguity and re-write the grammar.

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be *ambiguous.*
- Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

Eliminating Ambiguity

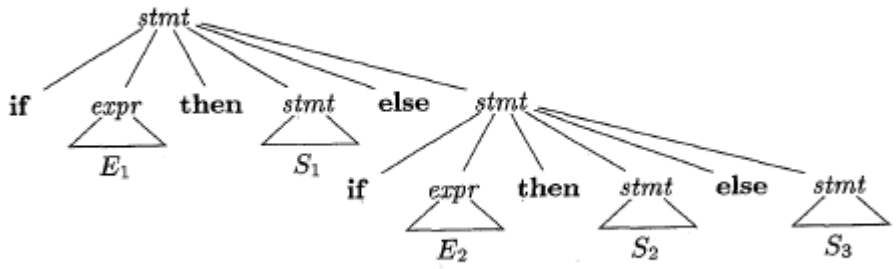
Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.

As an example, we shall eliminate the ambiguity from the following "dangling else" grammar:

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

Here **"other"** stands for any other statement. According to this grammar, the compound conditional statement

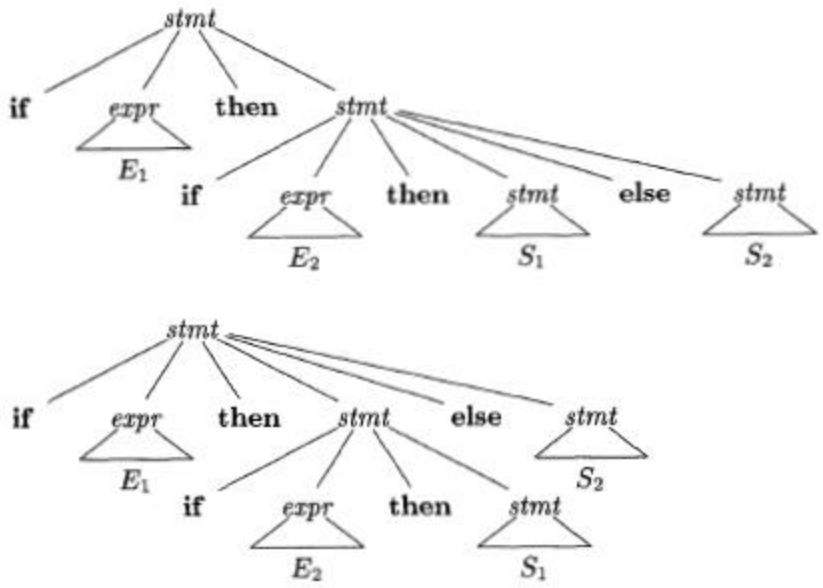
$$\textbf{if } E_1 \textbf{ then } S_1 \textbf{ else if } E_2 \textbf{ then } S_2 \textbf{ else } S_3$$



Parse tree for a conditional statement

Above grammar is ambiguous since the string

$\textbf{if } E_1 \textbf{ then } S_1 \textbf{ else if } E_2 \textbf{ then } S_2 \textbf{ else } S_3$ Have the two parse trees below;



Two parse trees for an ambiguous sentence

stmt	→	*matched_stmt*
	|	*open_stmt*
matched_stmt	→	**if** *expr* **then** *matched_stmt* **else** *matched_stmt*
	|	**other**
open_stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *matched_stmt* **else** *open_stmt*

Unambiguous grammar for if-then-else statements

Top Down Parsers

Left recursion

- *A* grammar is *left recursive* if it has a nonterminal A such that there is a $A \overset{+}{\Rightarrow} A\alpha$ for some string α
- Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.
- *Immediate left recursion,* where there is a production of the form $A \rightarrow A\alpha$.
- The left-recursive pair of productions $A \rightarrow A\alpha \mid \beta$ could be replaced by the non-left-recursive productions:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

without changing the strings derivable from A. This rule by itself suffices for many grammars.

- Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions. First, group the productions as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where no β_i begins with an A. Then, replace the A-productions by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

The nonterminal A generates the same strings as before but is no longer left recursive.

Algorithm : Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

```
1)  arrange the nonterminals in some order A1, A2, ..., An.
2)  for ( each i from 1 to n ) {
3)      for ( each j from 1 to i - 1 ) {
4)          replace each production of the form Ai → Aj γ by the
                productions Ai → δ1 γ | δ2 γ | ··· | δk γ, where
                Aj → δ1 | δ2 | ··· | δk are all current Aj-productions
5)      }
6)      eliminate the immediate left recursion among the Ai-productions
7)  }
```

Algorithm to eliminate left recursion from a grammar

For example, consider the grammar

$$S \rightarrow A\ a \mid b$$
$$A \rightarrow A\ c \mid S\ d \mid \epsilon$$

The nonterminal S is left recursive because $S \Rightarrow Aa \Rightarrow Sda$, but it is not immediately left recursive.

We order the nonterminals S, A. There is no immediate left recursion among the S-productions, so nothing happens during the outer loop for $i = 1$. For $i = 2$, we substitute for S in $A \rightarrow S\ d$ to obtain the following A-productions.

$$A \rightarrow A\ c \mid A\ a\ d \mid b\ d \mid \epsilon$$

Eliminating the immediate left recursion among these A-productions yields the following grammar.

$$S \rightarrow A\ a \mid b$$
$$A \rightarrow b\ d\ A' \mid A'$$
$$A' \rightarrow c\ A' \mid a\ d\ A' \mid \epsilon$$

Grammar for simple arithmetic expression

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

The non-left-recursive expression grammar is

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \textbf{id}
\end{array}
$$

Left factoring.

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing.
- When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

For example, if we have the two productions

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
 & \mid & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}
\end{array}
$$

on seeing the input **if**, we cannot immediately tell which production to choose to expand *stmt*. In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$. However, we may defer the decision by expanding A to $\alpha A'$. Then, after seeing the input derived from α, we expand A' to β_1 or to β_2. That is, left-factored, the original productions become

$$
\begin{array}{l}
A \rightarrow \alpha A' \\
A' \rightarrow \beta_1 \mid \beta_2
\end{array}
$$

Algorithm : Left factoring a grammar.

INPUT: Grammar G.

OUTPUT: An equivalent left-factored grammar.

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{aligned} A &\to \alpha A' \mid \gamma \\ A' &\to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. □

Example : The following grammar abstracts the "dangling-else" problem:

$$\begin{aligned} S &\to i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a \\ E &\to b \end{aligned}$$

Here, i, t, and e stand for **if**, **then**, and **else**; E and S stand for "conditional expression" and "statement." Left-factored, this grammar becomes:

$$\begin{aligned} S &\to i\ E\ t\ S\ S' \mid a \\ S' &\to e\ S \mid \epsilon \\ E &\to b \end{aligned}$$

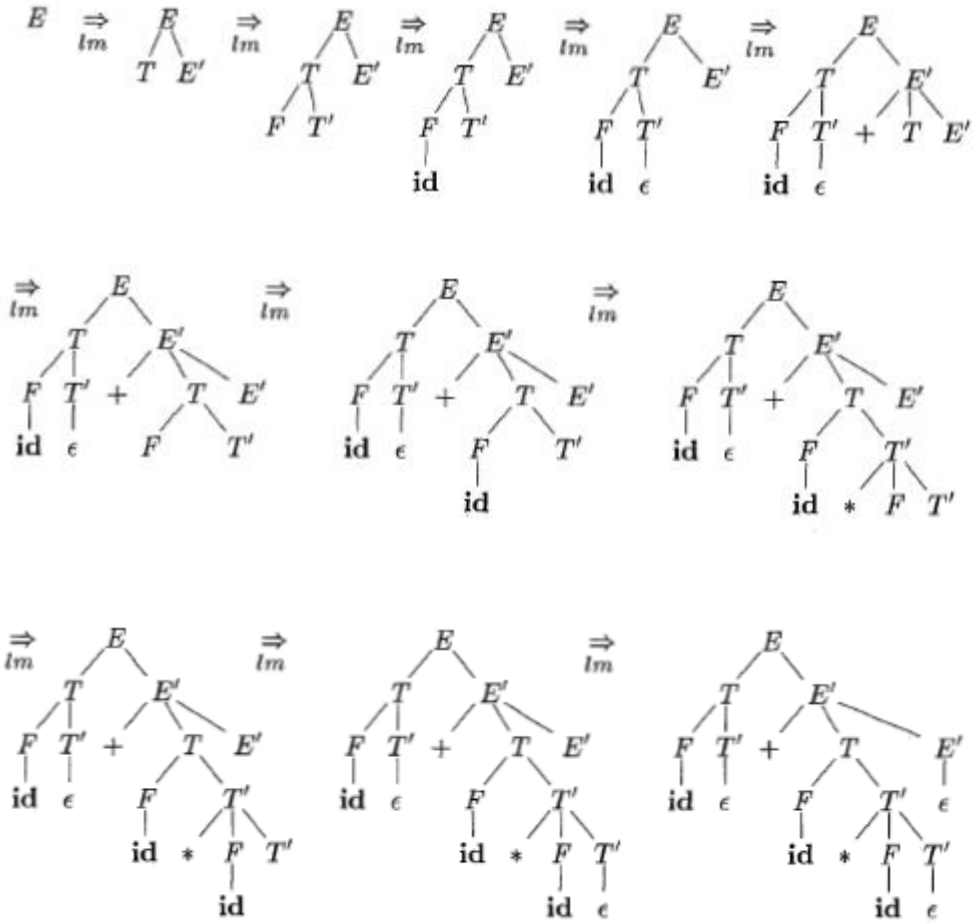
Thus, we may expand S to $iEtSS'$ on input i, and wait until $iEtS$ has been seen to decide whether to expand S' to eS or to ϵ. Of course, these grammars are both ambiguous, and on input e, it will not be clear which alternative for S' should be chosen.

Top-down parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first).
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

The sequence of parse trees in figure for the input **id + id * id** is a top-down parse according to below grammar repeated here:

$$\begin{aligned} E &\to T\ E' \\ E' &\to +\ T\ E' \mid \epsilon \\ T &\to F\ T' \\ T' &\to *\ F\ T' \mid \epsilon \\ F &\to (\ E\ ) \mid \textbf{id} \end{aligned}$$



Top-down parse for **id + id * id**

Recursive-Descent Parsing

```
     void A() {
1)       Choose an A-production, A → X1X2 ··· Xk;
2)       for ( i = 1 to k ) {
3)           if ( Xi is a nonterminal )
4)               call procedure Xi();
5)           else if ( Xi equals the current input symbol a )
6)               advance the input to the next symbol;
7)           else /* an error has occurred */;
         }
     }
```

A typical procedure for a nonterminal in a top-down parser

The following are the problems associated with top down parsing:

- Backtracking
- Left recursion
- Left factoring
- Ambiguity

Rules for constructing the FIRST and FOLLOW set.

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or **ϵ:** can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) = {X}.

2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST$(Y_1), \ldots,$ FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST$(Y_2)$, and so on.

3. If $X \to \epsilon$ is a production, then add ϵ to FIRST(X).

Now, we can compute FIRST for any string $X_1 X_2 \cdots X_n$ as follows. Add to FIRST$(X_1 X_2 \cdots X_n)$ all non-ϵ symbols of FIRST(X_1). Also add the non-ϵ symbols of FIRST(X_2), if ϵ is in FIRST(X_1); the non-ϵ symbols of FIRST(X_3), if ϵ is in FIRST(X_1) and FIRST(X_2); and so on. Finally, add ϵ to FIRST$(X_1 X_2 \cdots X_n)$ if, for all i, ϵ is in FIRST(X_i).

To compute FOLLOW (A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.

2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).

3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Consider again the non-left-recursive grammar

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

1. FIRST(F) = FIRST(T) = FIRST(E) = $\{(, \mathbf{id}\}$. To see why, note that the two productions for F have bodies that start with these two terminal symbols, **id** and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive ϵ, FIRST(T) must be the same as FIRST(F). The same argument covers FIRST(E).

2. FIRST$(E') = \{+, \epsilon\}$. The reason is that one of the two productions for E' has a body that begins with terminal $+$, and the other's body is ϵ. Whenever a nonterminal derives ϵ, we place ϵ in FIRST for that nonterminal.

3. FIRST$(T') = \{*, \epsilon\}$. The reasoning is analogous to that for FIRST(E').

4. FOLLOW(E) = FOLLOW(E') = $\{), \$\}$. Since E is the start symbol, FOLLOW(E) must contain \$. The production body $(\ E\ )$ explains why the right parenthesis is in FOLLOW(E). For E', note that this nonterminal appears only at the ends of bodies of E-productions. Thus, FOLLOW(E') must be the same as FOLLOW(E).

5. FOLLOW(T) = FOLLOW$(T') = \{+,), \$\}$. Notice that T appears in bodies only followed by E'. Thus, everything except ϵ that is in FIRST(E') must be in FOLLOW(T); that explains the symbol $+$. However, since FIRST(E') contains ϵ (i.e., $E' \stackrel{*}{\Rightarrow} \epsilon$), and E' is the entire string following T in the bodies of the E-productions, everything in FOLLOW(E) must also be in FOLLOW(T). That explains the symbols \$ and the right parenthesis. As for T', since it appears only at the ends of the T-productions, it must be that FOLLOW(T') = FOLLOW(T).

6. FOLLOW$(F) = \{+, *,), \$\}$. The reasoning is analogous to that for T in point (5).

Grammar G is LL (1) grammar

Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL (1). The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of look ahead at each step to make parsing action decisions.

The class of LL (1) grammars is rich enough to cover most programming constructs,

although care is needed in writing a suitable grammar for the source language. For example, no left-recursive or ambiguous grammar can be LL (1).

A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

Algorithm to construct predictive parsing table.

$E \to E + T \mid T$
$T \to T^* F \mid F$
$F \to (E) \mid id$

Algorithm : Construction of a predictive parsing table.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

- Remove Left recursion
- Find FIRST and FOLLOW
- Construct LL(1) table

Remove Left recursion

Grammar after removing left recursion is as follows.

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \textbf{id}
\end{array}
$$

Find FIRST and FOLLOW

FIRST(E)=FIRST(T)=FIRST(F)={(,id}
FIRST(E`)={+, _}
FIRST(T`)={*, _}
FOllOW(E)=FOLLOW(E`)={),$}
FOllOW(T)=FOLLOW(T`)={+,),$}
FOLLOW(F)={*,+,),$}

Construct LL (1) table

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \textbf{id}$			$F \rightarrow (E)$		

Parsing table M

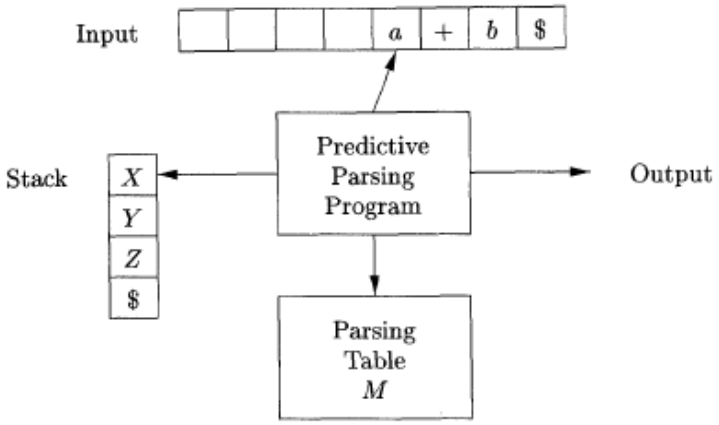
Moves made by a predictive parser on input id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	**id + id * id$**	
	$TE'\$$	**id + id * id$**	output $E \rightarrow TE'$
	$FT'E'\$$	**id + id * id$**	output $T \rightarrow FT'$
	id $T'E'\$$	**id + id * id$**	output $F \rightarrow$ **id**
id	$T'E'\$$	**+ id * id$**	match **id**
id	$E'\$$	**+ id * id$**	output $T' \rightarrow \epsilon$
id	$+\ TE'\$$	**+ id * id$**	output $E' \rightarrow +\ TE'$
id +	$TE'\$$	**id * id$**	match +
id +	$FT'E'\$$	**id * id$**	output $T \rightarrow FT'$
id +	**id** $T'E'\$$	**id * id$**	output $F \rightarrow$ **id**
id + id	$T'E'\$$	*** id$**	match **id**
id + id	$*\ FT'E'\$$	*** id$**	output $T' \rightarrow *\ FT'$
id + id *	$FT'E'\$$	**id$**	match *
id + id *	**id** $T'E'\$$	**id$**	output $F \rightarrow$ **id**
id + id * id	$T'E'\$$	**$**	match **id**
id + id * id	$E'\$$	**$**	output $T' \rightarrow \epsilon$
id + id * id	**$**	**$**	output $E' \rightarrow \epsilon$

Moves made by a predictive parser on input **id + id * id**

Table driven predictive parser (Non-recursive Predictive Parsing).

- It is possible to build a non-recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls.
- The parser mimics a leftmost derivation.
- The non-recursive parser looks up the production to be applied in a parsing table.
- The table can be constructed directly from LL (1) grammars.



- **An input buffer**
  - Contains the input string.
  - The string can be followed by $, an end marker to indicate the end of the string.
- **A stack**
  - Contains symbols with $ on the bottom, with the start symbol initially on the top.
- **A parsing table** (2-dimensional array M [A, a]).
- **An output stream** (production rules applied for derivation).

Algorithm : Table-driven predictive parsing.

INPUT: A string w and a parsing table M for grammar G.

OUTPUT: If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input.

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X is a ) pop the stack and advance ip;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1 Y2 ··· Yk ) {
                output the production X → Y1 Y2 ··· Yk;
                pop the stack;
                push Yk, Yk−1, ... , Y1 onto the stack, with Y1 on top;
        }
        set X to the top stack symbol;
}
```

Predictive parsing algorithm

The different Error Recovery in Predictive Parsing.

An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and *M[A, a]* is **error** (i.e., the parsing-table entry is empty).

1. **Panic Mode**
2. **Phrase-level Recovery**

Panic mode error recovery: It is based on the idea of skipping over symbols on the input until a token in a selected set of synchronizing tokens appears. Place all symbols in FOLLOW (A) into the synchronizing set for nonterminal A. if a terminal on the top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing.

Example

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$	synch	synch	$F \to (E)$	synch	synch

Figure : Synchronizing tokens added to the parsing table of Fig. 4.17

On the erroneous input) **id** $*$ $+$ **id**, the parser and error recovery mechanism

STACK	INPUT	REMARK
E \$	) **id** $*$ $+$ **id** \$	error, skip)
E \$	**id** $*$ $+$ **id** \$	**id** is in FIRST(E)
TE' \$	**id** $*$ $+$ **id** \$	
$FT'E'$ \$	**id** $*$ $+$ **id** \$	
id $T'E'$\$	**id** $*$ $+$ **id** \$	
$T'E'$ \$	$*$ $+$ **id** \$	
$*FT'E'$ \$	$*$ $+$ **id** \$	
$FT'E'$ \$	$+$ **id** \$	error, $M[F,+]$ = synch
$T'E'$ \$	$+$ **id** \$	F has been popped
E' \$	$+$ **id** \$	
$+TE'$ \$	$+$ **id** \$	
TE' \$	**id** \$	
$FT'E'$ \$	**id** \$	
id $T'E'$ \$	**id** \$	
$T'E'$ \$	\$	
E' \$	\$	
\$	\$	

Parsing and error recovery moves made by a predictive parser

Phrase level recovery

Phrase-level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack. Alteration of stack symbols or the pushing of new symbols onto the stack is questionable for several reasons. First, the steps carried out by the parser might then not correspond to the derivation of any word in the language at all. Second, we must ensure that there is no possibility of an infinite loop. Checking that any recovery action eventually results in an input symbol being consumed (or the stack being

shortened if the end of the input has been reached) is a good way to protect against such loops.

University Questions/Question bank:

1. Define CFG. Write a CFG to specify

   i) All string over {a, b} that are even and odd palindromes.

   ii) L = { $a^n b^{2n}$ over Σ = {a, b} $n \geq 1$ }

2. Define CFG. Design CFG for the languages

   i) L={02^n1^m | n>=0,m>=0 }

   ii) L={$0^i 1^j 2^k$ |i=j or j=k}

3 Define Ambiguity. Consider the grammar E->E+E|E*E|(E)|id. Find the leftmost, rightmost derivations and parse trees for the string id+id*id. And show that this grammar is ambiguous.

4, Let G be the grammer,

S→aB | bA

A→a | aS | bAA

B→b | bS | aBB

For the string aaabbabbba find a.

(i) Left most derivation

(ii) Right most derivation

(iii)Parse tree

5 a. Define context free grammer. Design a context free grammer for the languages.

I. L={$0^m 1^m 2^n$ | $m \geq 0, n \geq 0$}

II. L={ $a^i b^j$ |$i \neq j$,$i \geq 0, j \geq 0$}

III. L={$a^n b^{n-3}$ | $n \geq 3$}.

b. Consider the grammer G with production.

S -> AbB

A -> aA|€

B-> aB|bB|€

Obtain leftmost derivation,rightmost dervation and parse tree for the string aaabab.

6. Obtain the grammar to generate the language L = {w|$n_a(w) = n_b(w)$}|.

7. Show that the following grammar is ambiguous.

S-> aB|bA

A->aS|bAA|a

B-> bS|aBB|b.

8 Write grammar for the following languages:

i)L ={0^{n+2} 1^n| n>=1}

ii)L ={$a^n b^m$|m>n and n>=0}

9.Consider the grammar G, with productions:

S -> AB | aaB

A -> aA |∈

B -> aB| bB|∈

Give leftmost derivation, right most derivation and parse tree for the string aaabab

10.show that the following grammer is ambiguous.

S->AB | aaB

A->a|Aa

11 Define CFG. Design a context free grammar for the languages:

(i) L={a^i b ^j ^c, where i= j+k ,i j K≥0)

(ii) L={ 0 ^n+2 1 ^n : n≥1}

12 what is ambiguous grammar ? Show that the grammar shown below is a ambiguous on the string "abb"

(i). S-> AB/aaB

A-> Aa/a

B->b

13 Consider the grammar : E->+ EE/ * EE/- EE/X/Y

Find the left most deviation ,right Most deviation and parse tree for the string "+ *xy xy".

14 a.Obtain the CFG's for the following languages :

i) L_1={ $a^R ww^R b^n$|w ∈{0,1}* and n >=2}

ii) L_2={ $a^k b^m c^n$|m + n=k and m,n>=1}

iii) L_3={w ∈ {a}*| |w|mod 3≠|w| mod 2}

b. Consider the CFG with productions

E→ E * T|T

T→F – T |F

F→(E) | 0 |1

Write the leftmost derivation , rightmost derivation and parse tree for the string

*0 –((1*0)-0)*

c. Show that the following grammar is ambiguous :

S→SbS

S→a

15. Write CFG for the following languages.

i) L = {0n1n | n ≥ 1 }

ii) L = {String l of a's and b's with equal number of a's and b's }

16. Show that the following grammar is ambiguous

E → E + E| E – E | E * E | E / E | (E) | a

Where E is the start symbol. Find the unambigous grammar.

17. What is Syntax analysis? With a neat diagram explain the role of parser.
18. Explain different error recovery strategies.
19. Give the formal definition of CFG with an example.
20. What is derivation? Consider the following context free grammar
21. $E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$ and the input string – (id + id).
22. Give LMD and RMD
23. Parse tree
24. Is the grammar ambiguous? why
25. Why it is necessary for regular expressions to define the lexical syntax of a language? Give Reasons
26. Define Ambiguity. Is the following grammar ambiguous? If yes remove the ambiguity and re-write the grammar.

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & \mid & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & \mid & \textbf{other}
\end{array}
$$

27.

28. What is left recursion? How left recursion can be eliminated from the grammars? Write down the simple arithmetic expression grammar and re-write the grammar after eliminating the left recursion.
29. What is left factoring? How left factoring can be eliminated from the grammars? Explain with suitable example.

30. What is top-down parsing? What are the problems associated with the top-down parsing? Explain with suitable example.
31. Give the Rules for constructing the FIRST and FOLLOW set.
32. When we say that grammar G is LL (1) grammar?
33. Give an algorithm to construct predictive parsing table. Construct the predictive parsing the table by making the necessary changes to the following grammar and show the parsing of the string id + id *id.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T^* F \mid F$$
$$F \rightarrow (E) \mid id$$

34. Explain with a neat diagram, the model of a table driven predictive parser (Non-recursive Predictive Parsing).
35. List and explain the different Error Recovery in Predictive Parsing.