Unit - II

Chapter 4

Syntax Analysis

Top – Down Parsing

Outline

- Role of the parser
- Top-Down parsing:
  - Predictive Parsing
    - Recursive, and
    - Nonrecursive

Introduction

- The syntax of the programming language constructs can be described by context free grammars or BNF (Backnus-Naur Form).
- Grammar offers significant advantage to both language designer and compiler writers.
- A grammar gives precise, yet easy-to understand, syntactic specification of a programming language.
- From certain class of grammars we can automatically construct an efficient parser that determines if a source program is syntactically well formed.
- A properly designed grammar imparts a structure to a programming language that is useful for the translation of source program into correct object code and for the detection of errors.
- Languages evolve over a period of time, acquiring new constructs and performing additional tasks.

Backus-Naur Form (BNF)

- Backus-Naur form (BNF) is a formal notation for encoding grammars intended for human consumption.
- Many programming languages, protocols or formats have a BNF description in their specification.
- Every rule in Backus-Naur form has the following structure:

name ::= expansion

- The symbol '::=' means "may expand into" and "may be replaced with."
- a name is also called a non-terminal symbol.

Backus-Naur Form (BNF)

- Every name in Backus-Naur form is surrounded by angle brackets, < >, whether it appears on the left- or right-hand side of the rule.
- An expansion is an expression containing terminal symbols and non-terminal symbols, joined together by sequencing and choice.
- A terminal symbol is a literal like ("+" or "function") or a class of literals (like integer).
- Simply juxtaposing expressions indicates sequencing.
- A vertical bar ‘|’ indicates choice.

Backus-Naur Form (BNF)

- For example, in BNF, the classic expression grammar is:

<expr> ::= <term> "+" <expr>

| <term>

<term> ::= <factor> "*" <term>

| <factor>

<factor> ::= "(" <expr> ")“

| <const>

<const> ::= integer

Backus-Naur Form (BNF)

- Naturally, we can define a grammar for rules in BNF:

  rule $\rightarrow$ name ::= expansion

  name $\rightarrow$ < identifier >

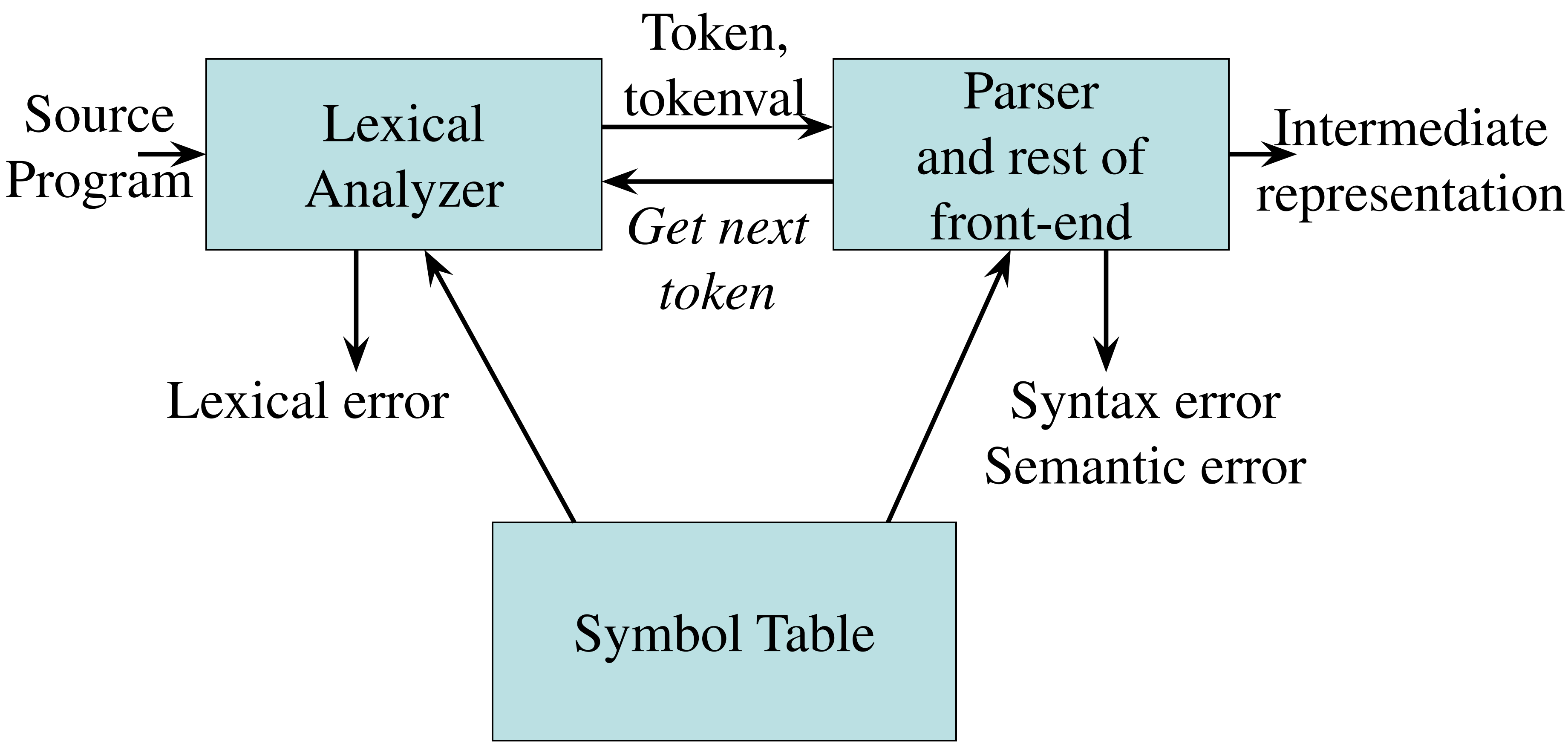
  expansion $\rightarrow$ expansion expansion

  expansion $\rightarrow$ expansion | expansion

  expansion $\rightarrow$ name

  expansion $\rightarrow$ terminal

Position of a Parser in the Compiler Model



The Parser

- The task of the parser is to check syntax
- The syntax-directed translation stage in the compiler's front-end checks static semantics and produces an intermediate representation (IR) of the source program
  - Abstract syntax trees (ASTs)
  - Control-flow graphs (CFGs) with triples, three-address code, or register transfer lists
  - WHIRL (SGI Pro64 compiler) has 5 IR levels!

Error Handling

- A good compiler should assist in identifying and locating errors
  - *Lexical errors*: important, compiler can easily recover and continue
    - Example: misspelling identifier, keyword or operator
  - *Syntax errors*: most important for compiler, can almost always recover
    - Example: an arithmetic expression with unbalanced parenthesis
  - *Static semantic errors*: important, can sometimes recover
  - *Dynamic semantic errors*: hard or impossible to detect at compile time, runtime checks are required
    - Example for semantic error: an operator applied to an incompatible operand.
  - *Logical errors*: hard or impossible to detect
    - Example: an infinitely recursive call.

Error Handling

- The error handler in a parser 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 the processing of correct programs.

Viable-Prefix Property

- The *viable-prefix property* of LL/LR parsers allows early detection of syntax errors
  - Goal: detection of an error as soon as possible without consuming unnecessary input
  - How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language

Error is detected here

Prefix

```
…
for (;)
…
```

Error is detected here

Prefix

```
…
DO 10 I = 1;0
…
```

Error Recovery Strategies

- *Panic mode*
  - Discard input until a token in a set of designated synchronizing tokens is found
- *Phrase-level recovery*
  - Perform local correction on the input to repair the error
- *Error productions*
  - Augment grammar with productions for erroneous constructs
- *Global correction*
  - Choose a minimal sequence of changes to obtain a global least-cost correction

Grammars

- Context-free 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 is a designated *start symbol* $S \in N$

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

Derivation (Example)

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

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

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

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

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

Chomsky Hierarchy: Language Classification

- A grammar G is said to be
  - *Regular* if it is *right linear* where each production is of the form
    $A \rightarrow w\, B$ or $A \rightarrow w$
    or *left linear* where each production is of the form
    $A \rightarrow B\, w$ or $A \rightarrow w$
  - *Context free* if each production is of the form
    $A \rightarrow \alpha$
    where $A \in N$ and $\alpha \in (N \cup T)^*$
  - *Context sensitive* if each production is of the form
    $\alpha\, A\, \beta \rightarrow \alpha\, \gamma\, \beta$
    where $A \in N$, $\alpha,\gamma,\beta \in (N \cup T)^*$, $|\gamma| > 0$
  - *Unrestricted*

Chomsky Hierarchy

$\mathsf{L}(\textit{regular}) \subseteq \mathsf{L}(\textit{context free}) \subseteq \mathsf{L}(\textit{context sensitive}) \subseteq \mathsf{L}(\textit{unrestricted})$

Where $\mathsf{L}(T) = \{ L(G) \mid G \text{ is of type } T \}$
That is, the set of all languages
generated by grammars G of type T

Examples:

Every *finite language* is regular

$L_1 = \{ \mathbf{a}^n\mathbf{b}^n \mid n \geq 1 \}$ is context free

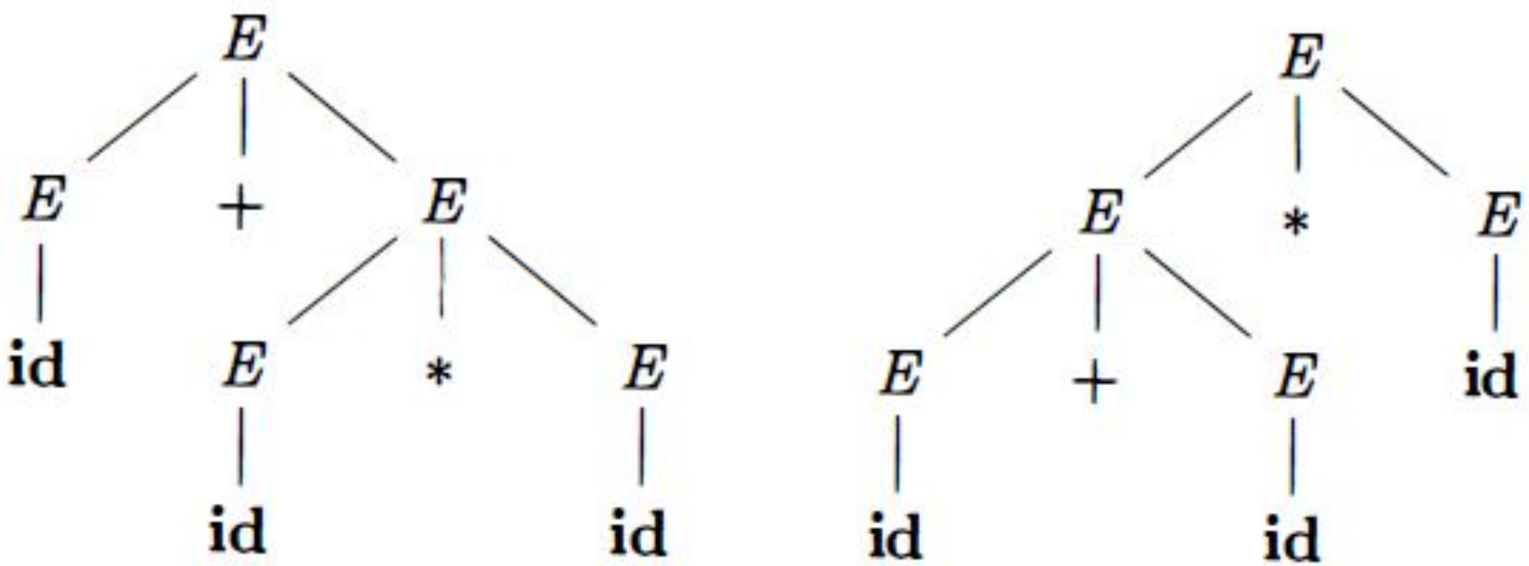
$L_2 = \{ \mathbf{a}^n\mathbf{b}^n\mathbf{c}^n \mid n \geq 1 \}$ is context sensitive

Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be ambiguous.
- Example: id + id * id
- Two distinct left derivation

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

Ambiguity: Two Parse trees



Eliminating Ambiguity

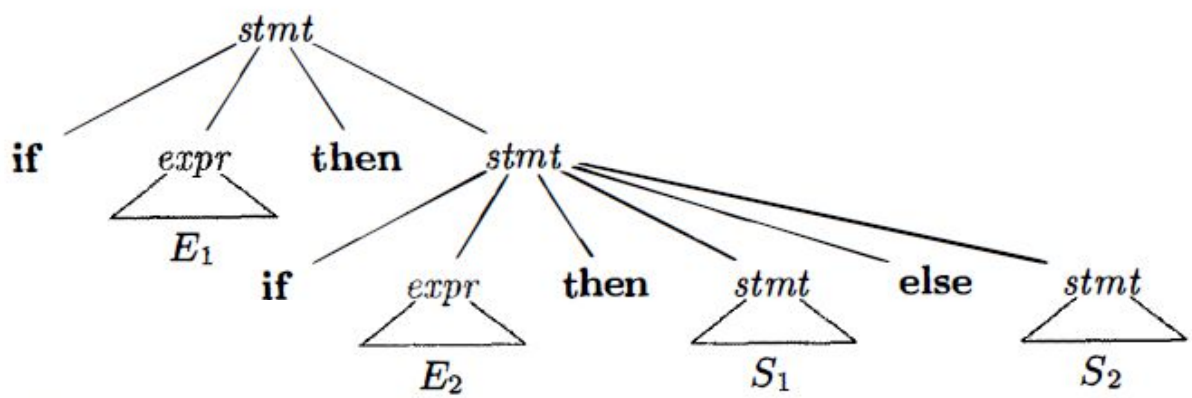
- “Dangling-else” grammar

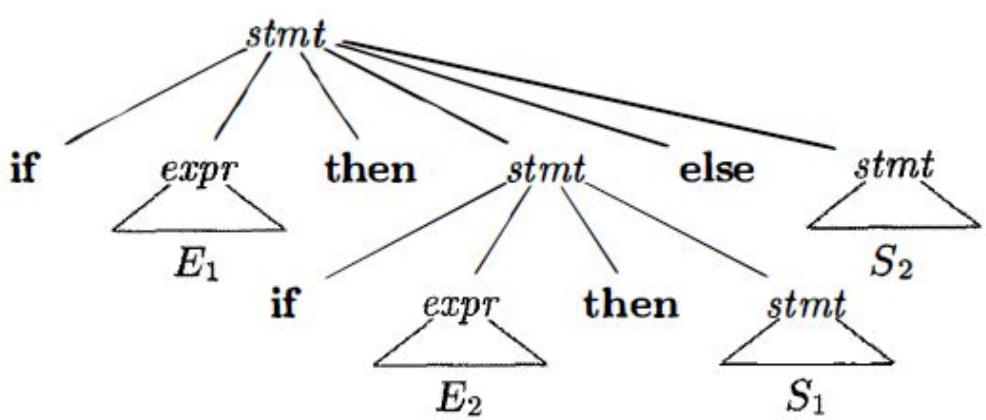
$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{other} \end{array}$$

- Grammar is ambiguous since the string

  if E_1 **then if** E_2 **then** S_1 **else** S_2

  has two parse trees





SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

Eliminating Ambiguity

- The general rule is, “Match each **else** with the closet previous unmatched **then**”.
- The idea is that a statement appearing between a **then** and an **else** must be "matched" ; that is, the interior statement must not end with an unmatched or open then. A matched statement is either an if-then-else statement containing no open statements or it is any other kind of unconditional statement.
- Now, the grammar, rewritten

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

Left Recursion

- Productions of the form
  $A \rightarrow A\ \alpha \mid \beta$
  are left recursive
- Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed.

Immediate Left-Recursion Elimination

Rewrite every left-recursive production

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

into a right-recursive production:

$A \rightarrow \beta\ A'$

$A' \rightarrow \alpha\ A' \mid \in$

Example

- Consider the grammar

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

  $F \rightarrow (E) \mid \textbf{id}$

- Eliminate immediate left recursion (Non-terminal E and T having such productions $A \rightarrow A\ \alpha$)

  $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \in$

  $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \in$

  $F \rightarrow (E) \mid \textbf{id}$

Another Example

- Consider the grammar, but it is not immediately left recursive.

  $S \rightarrow Aa \mid b$

  $A \rightarrow Ac \mid Sd \mid \in$

- Using general left recursion algorithm
- Substitute S-productions $A \rightarrow Sd$ to obtain the following productions

  $A \rightarrow Ac \mid Aad \mid bd \mid \in$

- Now, Eliminate the immediate left recursion among the A-productions

  $S \rightarrow Aa \mid b$

  $A \rightarrow bdA' \mid A'$

  $A' \rightarrow cA' \mid adA' \mid \in$

General Left Recursion Elimination

Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$

for $i = 1, \ldots, n$ **do**

    for $j = 1, \ldots, i\text{-}1$ **do**

        replace each

$$A_i \rightarrow A_j\,\gamma$$

        with

$$A_i \rightarrow \delta_1\,\gamma \mid \delta_2\,\gamma \mid \ldots \mid \delta_k\,\gamma$$

        where

$$A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$$

    enddo

    eliminate the immediate left recursion in A_i

enddo

Example Left Rec. Elimination

$$\left.\begin{aligned} A &\rightarrow B\ C \mid \mathbf{a} \\ B &\rightarrow C\ A \mid A\ \mathbf{b} \\ C &\rightarrow A\ B \mid C\ C \mid \mathbf{a} \end{aligned}\right\} \text{Choose arrangement: } A, B, C$$

$i = 1$: nothing to do

$i = 2, j = 1$: $B \rightarrow C\ A \mid \underline{A}\ \mathbf{b}$

$\Rightarrow\ B \rightarrow C\ A \mid \underline{B\ C}\ \mathbf{b} \mid \underline{\mathbf{a}}\ \mathbf{b}$

$\Rightarrow_{\text{(imm)}}\ B \rightarrow C\ A\ B_R \mid \mathbf{a}\ \mathbf{b}\ B_R$

$B_R \rightarrow C\ \mathbf{b}\ B_R \mid \in$

$i = 3, j = 1$: $C \rightarrow \underline{A}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow\ C \rightarrow \underline{B\ C}\ B \mid \underline{\mathbf{a}}\ B \mid C\ C \mid \mathbf{a}$

$i = 3, j = 2$: $C \rightarrow \underline{B}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow\ C \rightarrow \underline{C\ A\ B_R}\ C\ B \mid \underline{\mathbf{a}\ \mathbf{b}\ B_R}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow_{\text{(imm)}}\ C \rightarrow \mathbf{a}\ \mathbf{b}\ B_R\ C\ B\ C_R \mid \mathbf{a}\ B\ C_R \mid \mathbf{a}\ C_R$

$C_R \rightarrow A\ B_R\ C\ B\ C_R \mid C\ C_R \mid \in$

Left Factoring

- When a nonterminal has two or more productions whose right-hand sides start with the same grammar symbols, the grammar is not LL(1) and cannot be used for predictive parsing
- If $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \gamma$ are productions
- After Left-Factored,

  $A \rightarrow \alpha A' \mid \gamma$

  $A' \rightarrow \beta_1 \mid \beta_2$

- In general, Replace productions

  $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \gamma$

  with

  $A \rightarrow \alpha A' \mid \gamma$

  $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Example

- Consider the grammar

  $S \rightarrow iEtS \mid iEtSeS \mid a$

  $E \rightarrow b$

- Left factored, this grammar becomes:

  $S \rightarrow iEtSS' \mid a$

  $S' \rightarrow eS \mid \in$

  $E \rightarrow b$

Parsing

- *Universal* (any C-F grammar)
  - Cocke-Younger-Kasimi
  - Earley
- *Top-down* (C-F grammar with restrictions)
  - Recursive descent (predictive parsing)
  - LL (Left-to-right, Leftmost derivation) methods
- *Bottom-up* (C-F grammar with restrictions)
  - Operator precedence parsing
  - LR (Left-to-right, Rightmost derivation) methods
    - SLR, canonical LR, LALR

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Top-Down Parsing

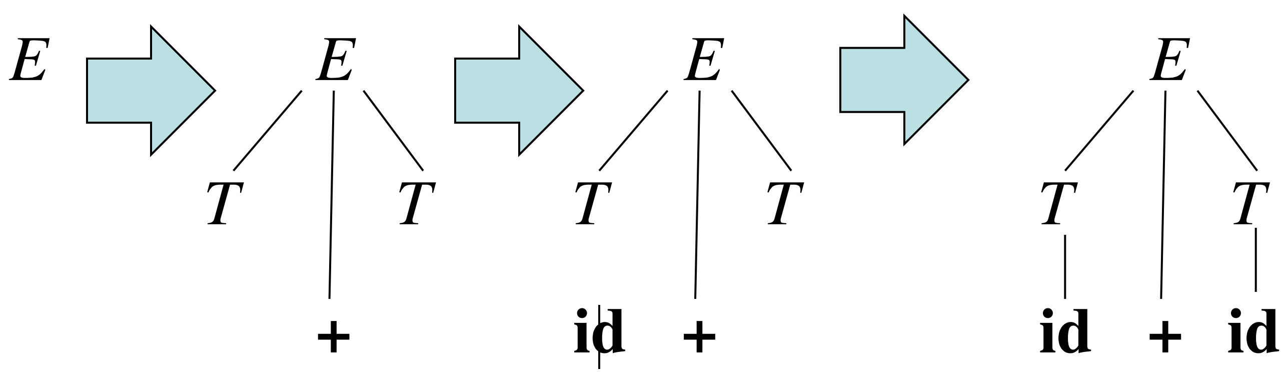
- LL methods (Left-to-right, Leftmost derivation) and recursive-descent parsing

Grammar:

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

Leftmost derivation:

$$E \underset{lm}{\Rightarrow} T + T$$
$$\underset{lm}{\Rightarrow} \mathbf{id} + T$$
$$\underset{lm}{\Rightarrow} \mathbf{id} + \mathbf{id}$$



Recursive Descent Parsing

- Grammar must be LL(1)
- Every nonterminal has one (recursive) procedure responsible for parsing the nonterminal's syntactic category of input tokens
- When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information.
- It may involve backtracking i.e. making repeated scans of the input.
- **It is implemented as a mutual recursive suite of functions that descend through a parse tree for the string, and as such are called "recursive descent parsers".**

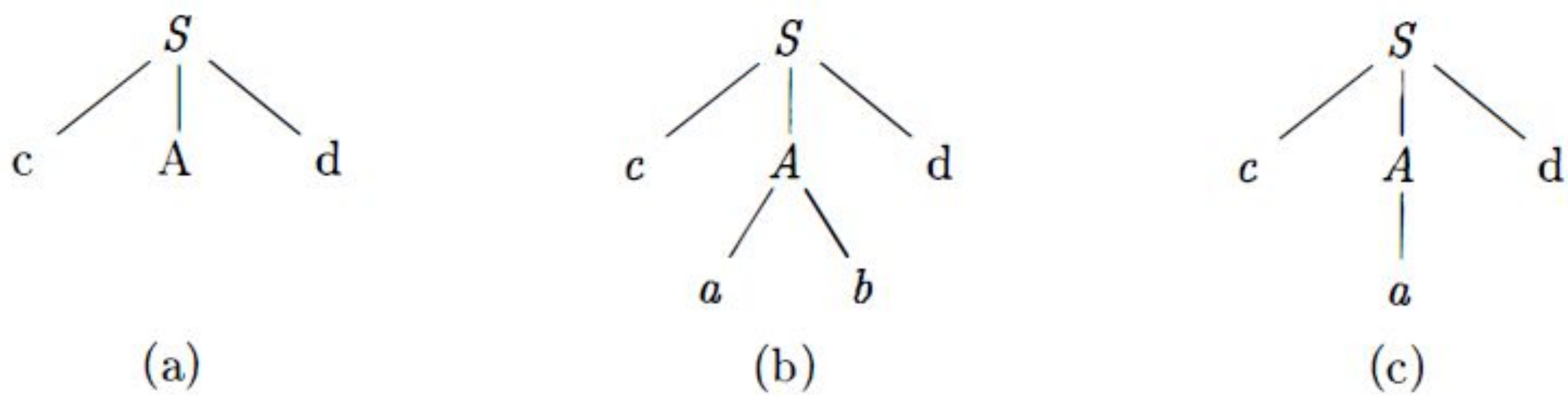
Example

- Consider the grammar

  S → cAd

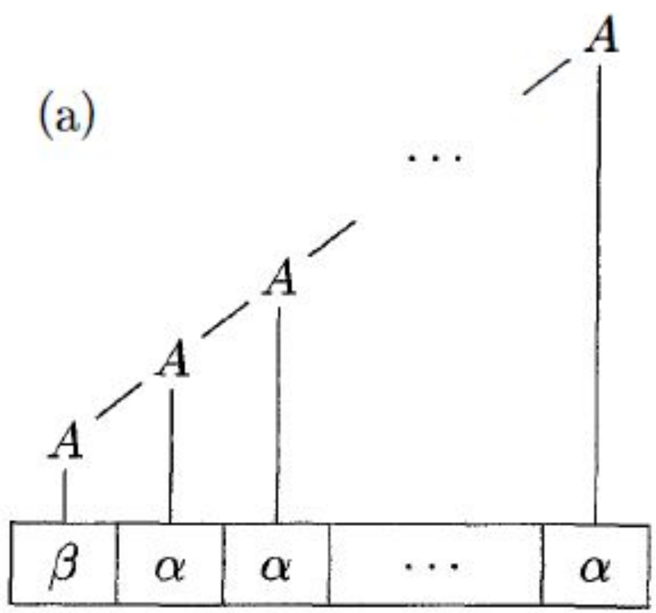
  A→ ab | a

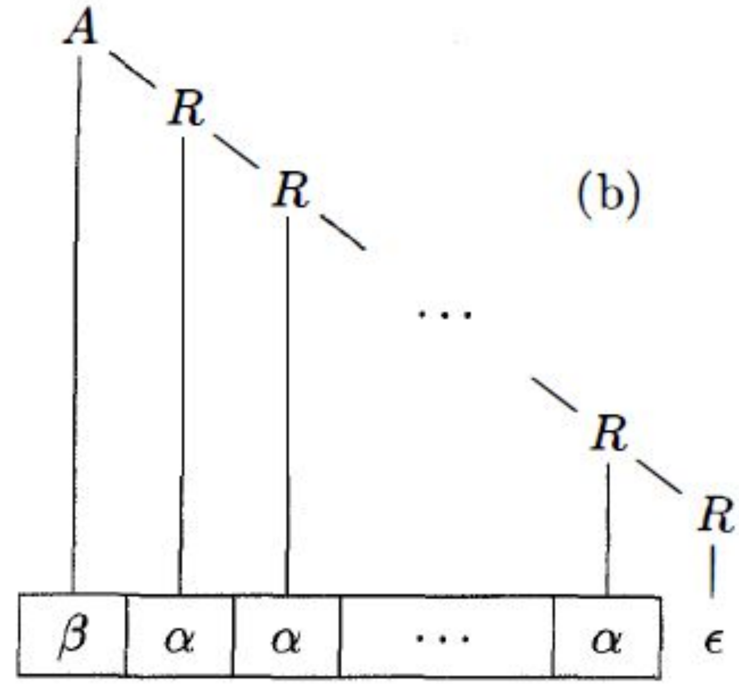
- Steps to build parse tree for string “**cad**”.



- **Note:** A left-recursive grammar can cause a recursive-decent parser, even one with backtracking, to go into an infinite loop i.e. try to expand A, it may eventually find ourselves again trying to expand A without having consumed any input.

It is possible for recursive-decent parser to loop forever





left-recursive production

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

right-recursive production:

$A \rightarrow \beta\ R$

$R \rightarrow \alpha\ R \mid \in$

Advantage and Limitations of recursive-descent parser

- Advantage:
  - It is simple to build.
  - It can be constructed with the help of parse tree.
- Limitations:
  - It is not very efficient as compared to other parsing techniques as there are chances that it may enter in an infinite loop for some input.
  - It is difficult to parse the string if lookahead symbol is arbitrarily long.

Predictive Parsing

- Eliminate left recursion from grammar
- Left factor the grammar
- Compute FIRST and FOLLOW
- Two variants:
  - Recursive (recursive calls)
  - Non-recursive (table-driven)

Transition Diagrams for Predictive Parsers

Consider the grammar:

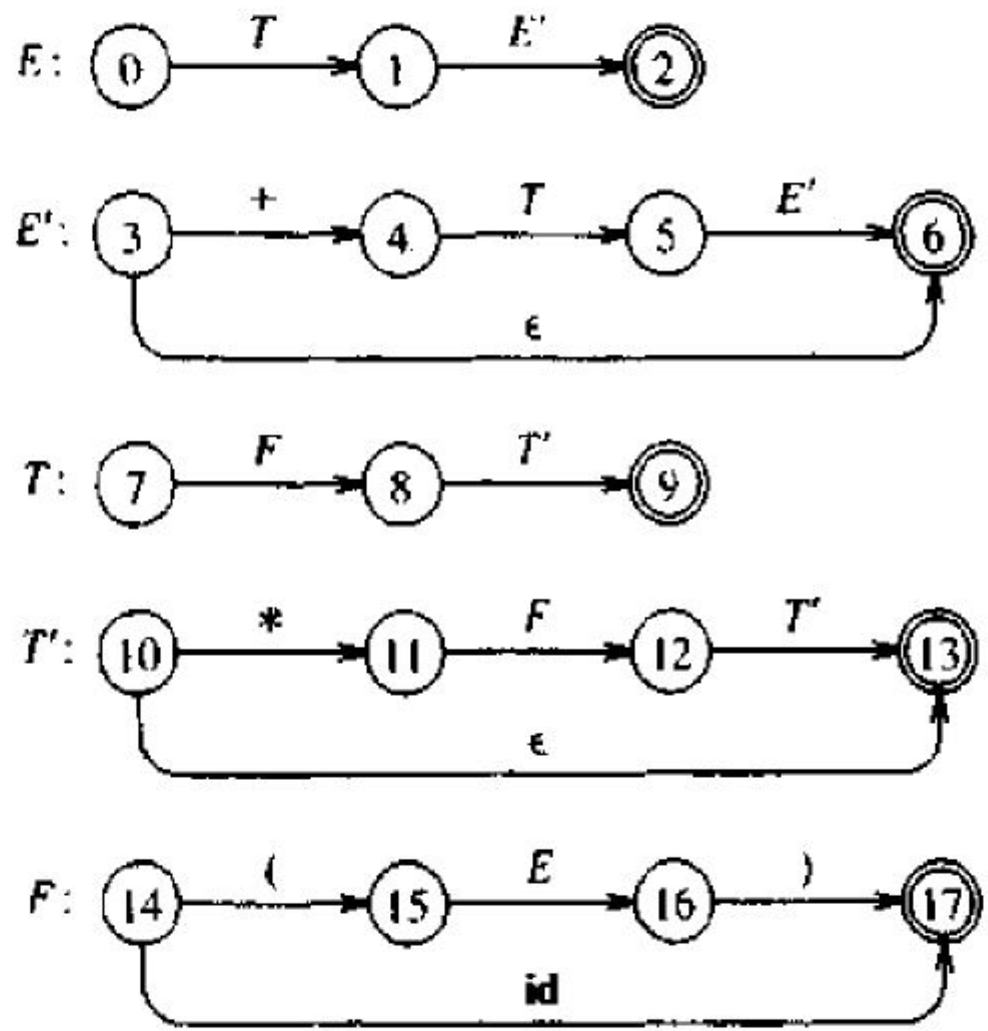
$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \in$

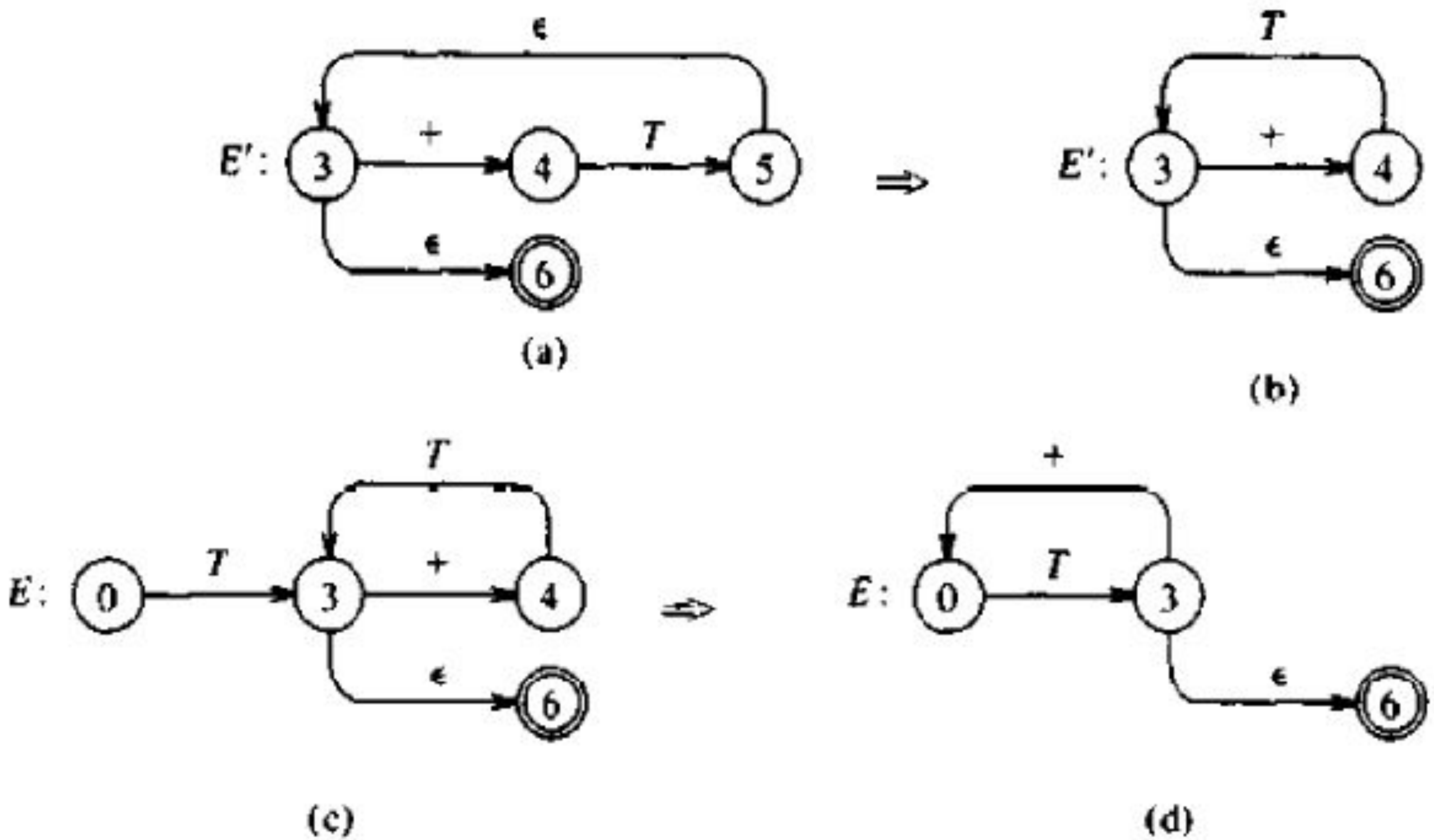
$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \in$

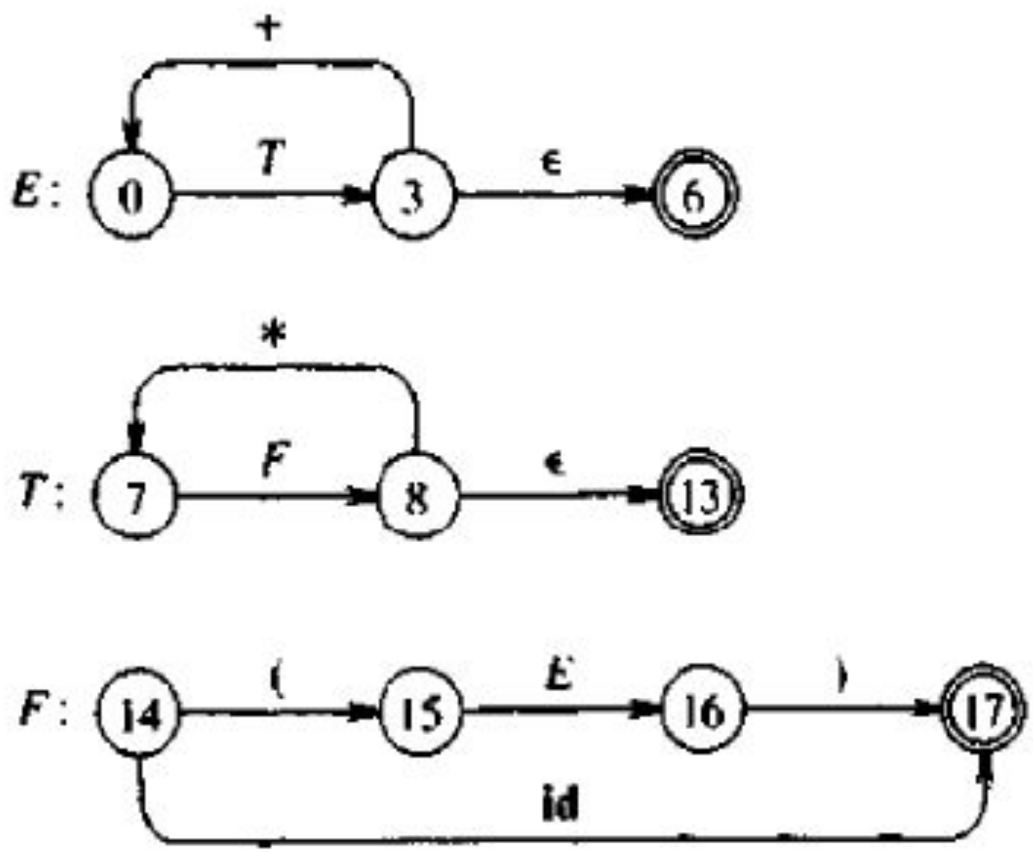
$F \rightarrow (E) \mid \mathbf{id}$



Transition Diagrams for Predictive Parsers



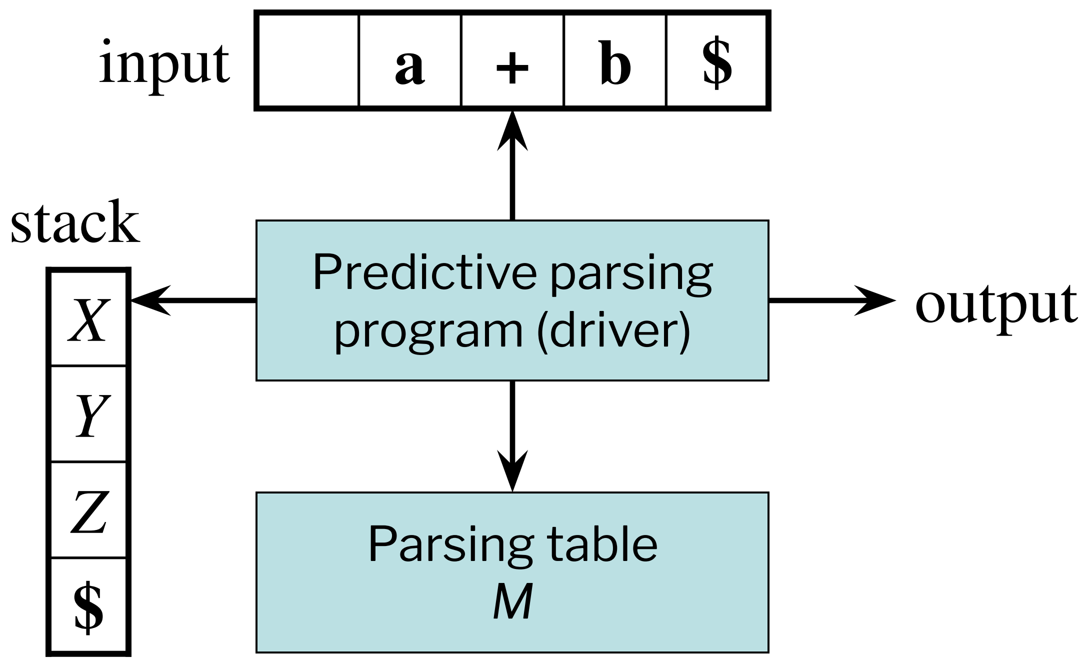
Transition Diagrams for Predictive Parsers



Simplified transition diagrams for arithmetic expressions.

Non-Recursive Predictive Parsing

- Given an LL(1) grammar $G=(N,T,P,S)$ construct a table $M[A,a]$ for $A \in N$, $a \in T$ and use a driver program with a stack



FIRST and FOLLOW

- **FIRST:** If α is any string of grammar symbols, let FIRST(α) be the set of terminals that begin the strings derived from α. If $\alpha \Rightarrow^* \in$, then $\in$ is also in FIRST(α).

- **FOLLOW:** it is defined as for nonterminal A i. e. FOLLOW(A), to be the set of terminals *a* that can appear immediately to the right of A in some sentential form, that is, the set of terminals *a* such that there exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$ for some α and β. If A is start symbol, then \$ is in FOLLOW(A).

FIRST

- To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or $\in$ can be added to any FIRST set.
  - If X is terminal, then FIRST(X) is {X}.
  - If $X \rightarrow \in$ is a production, then add $\in$ to FIRST(X).
  - If X is nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and $\in$ is in all of FIRST(Y_1), …, FIRST(Y_{i-1}); that is, $Y_1 \ldots Y_{i-1} \overset{*}{\Rightarrow} \in$. If $\in$ is in FIRST(Y_j) for all j=1, 2, …, k, then add $\in$ to FIRST(X).

Example

Consider the grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \in$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \in$

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

After applying FIRST rules over the grammar

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

FIRST(E') = { + , ∈ }

FIRST(T') = { * , ∈ }

FOLLOW

- To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.
  - Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.
  - If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for $\in$ is placed in FOLLOW(B).
  - If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains $\in$ (i.e. $\beta \overset{*}{\Rightarrow} \in$), then everything in FOLLOW(A) is in FOLLOW(B).

Example

Consider the grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \in$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \in$

$F \rightarrow (E) \mid \textbf{id}$

After applying FIRST rules over the grammar

FOLLOW(E) = FOLLOW(E') = {) , $ }

FOLLOW(T) = FOLLOW(T') = { + ,) , $ }

FOLLOW(F) = { + , * ,) , $ }

Usefulness of FIRST and FOLLOW

- FIRST and FOLLOW, both functions help for the construction of predictive parser, by fill in the entries of a predictive parsing table for grammar G, whenever possible.
- Sets of tokens yield by the FOLLOW function can also be used as synchronizing tokens during panic-mode error recovery.
- FIRST and FOLLOW also useful for LR parsing i. e. for LR(1) items and SLR(1) table.

Another Example of FIRST and FOLLOW

- Grammar G

  $S \rightarrow ACB \mid CbA \mid Ba$

  $A \rightarrow da \mid BC$

  $B \rightarrow g \mid \in$

  $C \rightarrow h \mid \in$

Nonterminal	FIRST	FOLLOW
S	d , g , h , ∈, b , a	$
A	d , ∈, g , h	h , g , $
B	g , ∈	$, a , h , g
C	h , ∈	g , $, b , h

Predictive Parsing Table

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

Construction of Predictive Parsing Table

Algorithm: Construction of a predictive parsing table.

Input: Grammar G.

Output: Parsing Table M.

Method:

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a].
3. If $\in$ is in FIRST(α), add $A \rightarrow \alpha$ to M[A, b] for each terminal b in FOLLOW(A). If $\in$ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, \$].
4. Make each undefined entry of M be **error**.

Predictive Parsing Working

- The program considers X, the symbols on the top of the stack, and a, the current input symbol. These two symbols determine the parser action.
- There are three possibilities:
  - If X = a = \$, the parser halts and announces successful completion of parsing.
  - If X = a ≠ \$, the parser pops X off the stack and advances the input pointer to the next input symbol.
  - If X is a nonterminal, the program consults entry M[X, a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If for example, M[X, a] = {X → UVW}, the parser replaces X on top of the stack by WVU (with U on top).
    - As output, we shall assume that the parser just prints the production used; any other code could be executed here.
    - If M[X, a] = error, the parser calls an error recovery routine.

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

Moves made by the Nonrecursive predictive parser

STACK	INPUT	OUTPUT
$\$E$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
$\$E'T$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$E \to TE'$
$\$E'T'F$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$T \to FT'$
$\$E'T'\mathbf{id}$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$F \to \mathbf{id}$
$\$E'T'$	$+ \mathbf{id} * \mathbf{id}\$$	
$\$E'$	$+ \mathbf{id} * \mathbf{id}\$$	$T' \to \epsilon$
$\$E'T+$	$+ \mathbf{id} * \mathbf{id}\$$	$E' \to +TE'$
$\$E'T$	$\mathbf{id} * \mathbf{id}\$$	
$\$E'T'F$	$\mathbf{id} * \mathbf{id}\$$	$T \to FT'$
$\$E'T'\mathbf{id}$	$\mathbf{id} * \mathbf{id}\$$	$F \to \mathbf{id}$
$\$E'T'$	$* \mathbf{id}\$$	
$\$E'T'F*$	$* \mathbf{id}\$$	$T' \to *FT'$
$\$E'T'F$	$\mathbf{id}\$$	
$\$E'T'\mathbf{id}$	$\mathbf{id}\$$	$F \to \mathbf{id}$
$\$E'T'$	$\$$	
$\$E'$	$\$$	$T' \to \epsilon$
$\$$	$\$$	$E' \to \epsilon$

LL(1) Grammar

- LL(1) means
  - The first "L": scanning the input from left to right.
  - The second "L": Leftmost derivation
  - "1" stands for Using one input symbol of lookahead at each step to make parsing action decisions.

Example of LL(1) Grammar that are ambiguous

- Consider the Grammar

  S → iEtSS' | *a*

  S'→ eS | ∈

  E → b

FIRST (S) = {*i*, *a*} FOLLOW(S) = { $, e }
FIRST (S') = {*e*, ∈ } FOLLOW(S') = { $, e }
FIRST (E) = { b } FOLLOW(E) = { t }

- Parsing Table for this grammar

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

- **A grammar whose parsing table has no multiply-defined entries is said to be LL(1).**

? What should be done when a parsing table has multiply-defined entries?

LL(1) Grammar Properties

- No ambiguous or Left recursive grammar can be LL(1).
- A grammar G is LL(1) iff whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G. The following conditions hold:
  - For no terminal *a*,

    do both α and β derive strings beginning with *a*.

    i.e. $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$
  - At most one of α and β can derive the empty string.

    i.e. if $\beta \Rightarrow^* \in$ then $\alpha \not\Rightarrow^* \in$ OR if $\alpha \Rightarrow^* \in$ then $\beta \not\Rightarrow^* \in$
  - If $\beta \Rightarrow^* \in$, then α does not derive any string beginning with a terminal in FOLLOW(A).

    i.e. if $\beta \Rightarrow^* \in$ then

    $\alpha \not\Rightarrow^* \in$

    $FIRST(\alpha) \cap FOLLOW(A) = \varnothing$

In General, LL(1) Grammar Properties

- A grammar G is LL(1) if for each collections of productions
 $A \longrightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
 for nonterminal A the following holds:

 1. $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \varnothing$ for all $i \neq j$
 2. if $\alpha_i \Rightarrow^* \in$ then
 2.a. $\alpha_j \Rightarrow^* \notin$ for all $i \neq j$
 2.b. $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \varnothing$
 for all $i \neq j$

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

Non-LL(1) Examples

Grammar	Not LL(1) because
$S \rightarrow S\,\mathbf{a} \mid \mathbf{a}$	Left recursive
$S \rightarrow \mathbf{a}\,S \mid \mathbf{a}$	FIRST($\mathbf{a}\,S$) $\cap$ FIRST($\mathbf{a}$) $\neq \varnothing$
$S \rightarrow \mathbf{a}\,R \mid \in$ $R \rightarrow S \mid \in$	For R: $S \Rightarrow^* \in$ and R $\Rightarrow^* \in$
$S \rightarrow \mathbf{a}\,R\,\mathbf{a}$ $R \rightarrow S \mid \in$	For R: FIRST(S) $\cap$ FOLLOW(R) $\neq \varnothing$
S $\rightarrow$ iEtSS' | a S' $\rightarrow$ eS | $\in$ E $\rightarrow$ b	Parsing table generate multiple defined entries.

Error Recovery in Predictive Parsing

- Two condition when error detected in predictive parsing.
  - When the terminal on top of the stack does not match the next input symbol.
  - When nonterminal A is on top of the stack, *a* is the next input symbol, and the parsing table entry M[A, *a*] is empty.
- Following error recovery method can be used.
  - Panic-mode error recovery
  - Phrase-level error recovery

Error Recovery in Predictive Parsing: Panic-mode

- It is based on the idea of ***skipping symbols on the input*** until a token in a selected set of ***synchronizing*** tokens appears.
- Its effectiveness depends on the choice of ***synchronizing set***.
- The set should be chosen so that the parser recovers quickly from errors that are ***likely to occur in practice***.

Error Recovery in Predictive Parsing: Panic-mode

- Rules
  - If the parser looks up entry **M[A, *a*] = blank**, then the input symbol is skipped.
  - If the entry is **synch**, then the nonterminal on top of the stack is popped in an attempt to resume parsing OR skip input until FIRST(A) found.
  - If a ***token on top of the stack*** does not match the input symbol, then we pop the token from the stack.

Error Recovery in Predictive Parsing: Panic-mode

- Add synchronizing actions to undefined entries based on FOLLOW.
- ***synch***: pop *A* and skip input till synch token OR skip until FIRST(*A*) found

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

Synchronizing tokens added to parsing table.

Error Recovery in Predictive Parsing: Panic-mode

- Erroneous input:) **id * + id**

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

Error Recovery in Predictive Parsing: Phrase-Level

- It 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.*
- In any case, it must be sure 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) . So, to protect against such loops.

Error Recovery in Predictive Parsing: Phrase-level

- Change input stream by inserting missing * For example: **id id** is changed into **id * id**

Nonterminal	INPUT SYMBOL					
	id	+	*	(	)	**$**
E	$E \rightarrow T E'$			$E \rightarrow T E'$	***synch***	***synch***
E'		$E' \rightarrow + T E'$			$E' \rightarrow \in$	$E' \rightarrow \in$
T	$T \rightarrow F T'$	***synch***		$T \rightarrow F T'$	***synch***	***Synch***
T'	***insert ****	$T' \rightarrow \in$	$T' \rightarrow * F T'$		$T' \rightarrow \in$	$T' \rightarrow \in$
F	$F \rightarrow$ **id**	***synch***	***synch***	$F \rightarrow (E)$	***synch***	***Synch***

insert *: insert missing * and redo the production

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

Error Recovery in Predictive Parsing: Phrase-level Error Productions

$E \rightarrow T\,E'$
$E' \rightarrow \mathbf{+}\,T\,E' \mid \in$
$T \rightarrow F\,T'$
$T' \rightarrow *\,F\,T' \mid \in$
$F \rightarrow (\,E\,) \mid \mathbf{id}$

Add error production:
$T' \rightarrow F\,T'$
to ignore missing *, e.g.: **id id**

Nonterminal	INPUT SYMBOL					
	id	+	*	(	)	**$**
E	$E \rightarrow T\,E'$			$E \rightarrow T\,E'$	***synch***	***synch***
E'		$E' \rightarrow +\,T\,E'$			$E' \rightarrow \in$	$E' \rightarrow \in$
T	$T \rightarrow F\,T'$	***synch***		$T \rightarrow F\,T'$	***synch***	***Synch***
T'	$T' \rightarrow F\,T'$	$T' \rightarrow \in$	$T' \rightarrow *\,F\,T'$		$T' \rightarrow \in$	$T' \rightarrow \in$
F	$F \rightarrow \mathbf{id}$	***synch***	***synch***	$F \rightarrow (\,E\,)$	***synch***	***Synch***

Error Recovery in Predictive Parsing: Phrase-level Error Productions

- Erroneous input: **id id**

STACK	INPUT	REMARKS
$E	id id $	
$ E' T	id id $	$E \rightarrow TE'$
$ E' T' F	id id $	$T \rightarrow FT'$
$ E' T' id	id id $	$F \rightarrow id$
$ E' T'	id $	**error**, ignore missing * , $M[T', id] = T' \rightarrow FT'$
$ E' T' F	id $	$T' \rightarrow FT'$
$ E' T' id	id $	$F \rightarrow id$
$ E' T'	$	
$ E'	$	$T' \rightarrow \epsilon$
$	$	$E' \rightarrow \epsilon$