

Compiler Design

Unit II

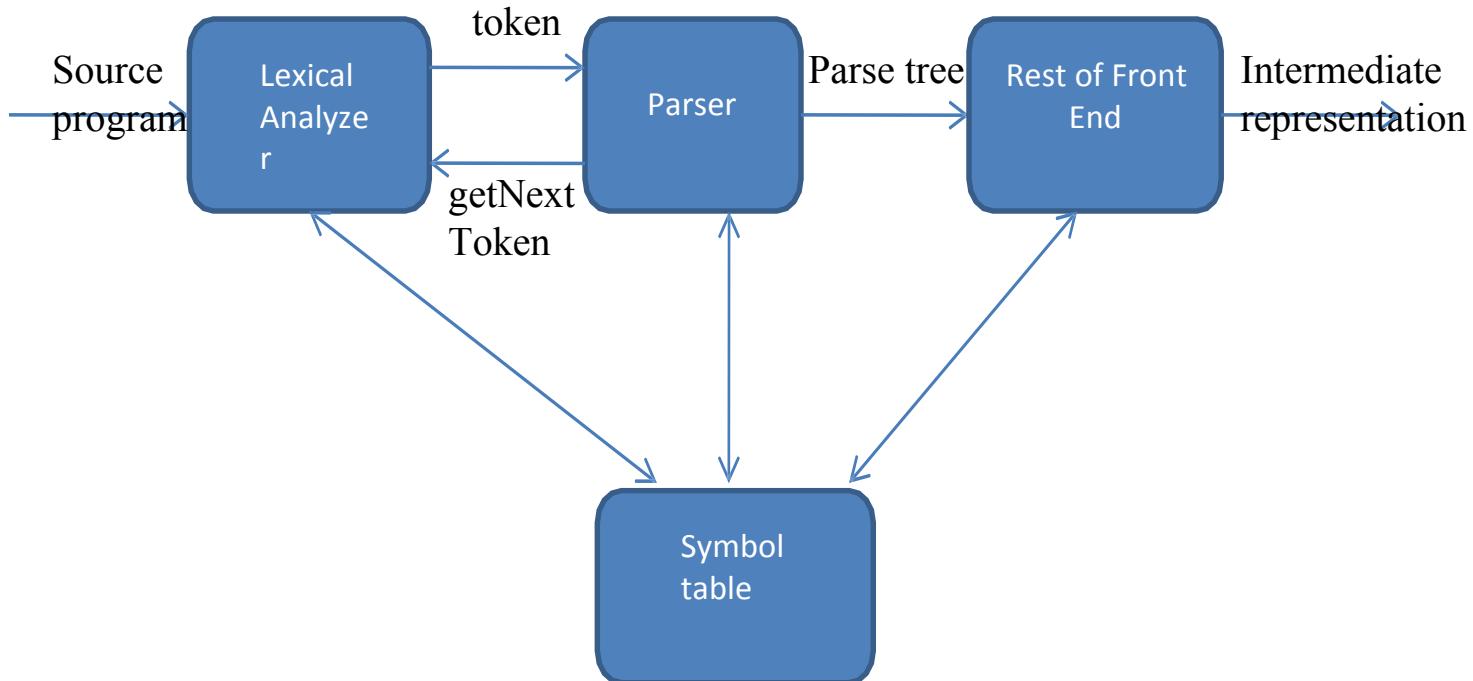
Syntax Analysis

- Definition - Role of parser
- Lexical versus Syntactic Analysis
- Representative Grammars
- Syntax Error Handling
- Elimination of Ambiguity, Left Recursion
- Left Factoring
- Top-down parsing
- Error Recovery in Predictive Parsing
- Predictive Parsing Algorithm
- Non Recursive Predictive Parser
- Recursive Descent Parsing, back tracking
- Computation of FIRST
- Problems related to FIRST
- Computation of FOLLOW
- Problems related to FOLLOW
- Construction of a predictive parsing table SLR
- Predictive Parsers LL(1) Grammars
- Transition Diagrams for Predictive Parsers

Parser

- Parser is a program that obtains tokens from lexical analyzer and constructs the parse tree which is passed to the next phase of compiler for further processing.
- Parser implements context free grammar for performing error checks

The role of parser



Syntax Analyzer

- *Syntax Analyzer* creates the syntactic structure of the given source program.
- This syntactic structure is mostly a *parse tree*.
- Syntax Analyzer is also known as *parser*.
- The syntax of a programming language is described by a *context-free grammar (CFG)*. We will use BNF (Backus-Naur Form) notation in the description of CFGs.
- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
 - If it satisfies, the parser creates the parse tree of that program.
 - Otherwise the parser gives the error messages.
- A context-free grammar
 - gives a precise syntactic specification of a programming language.
 - the design of the grammar is an initial phase of the design of a compiler.

Parsers (cont.)

- We categorize the parsers into two groups:
1. **Top-Down Parser**
 - the parse tree is created top to bottom, starting from the root.
 2. **Bottom-Up Parser**
 - the parse is created bottom to top; starting from the leaves
- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
 - Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
 - LL for top-down parsing
 - LR for bottom-up parsing

Lexical versus Syntactic Analysis

- **lexical analysis** reads the source code one character at a time and converts it into meaningful lexemes (tokens) whereas **syntax analysis** takes those tokens and produce a parse tree as an output
- **lexical analyzer** reads input program files and often includes buffering of that input, it is somewhat platform dependent. However, the **syntax analyzer** can be platform independent.
- **lexical analyzer** usually parses in terms of regular expressions, providing output that a **parser** uses **in the** form of tokens: identifiers, strings, numbers, operators. A **parser** implements a higher level grammar using the tokens as input.

Representative Grammars

- A context free grammar G is defined by four tuples as

$$G = (V, T, P, S)$$

where,

G – Grammar

V – Set of variables

T – Set of terminals

P – Set of productions

S – Start symbol

Representative Grammars

- **Terminals** are symbols from which strings are formed.
 - Lowercase letters, i.e., a , b , c .
 - Operators, i.e., $+$, $-$, $*$.
 - Punctuation symbols, i.e., comma, parenthesis.
 - Digits, i.e., 0 , 1 , 2 , \dots , 9 .
 - Boldface letters, i.e., **id**, **if**.
- **Non-terminals** are syntactic variables that denote a set of strings.
 - Uppercase letters, i.e., A , B , C .
 - Lowercase italic names, i.e., expr , stmt .
- **Start symbol** is the head of the production stated first in the grammar
- **Production** is of the form $\text{LHS} \rightarrow \text{RHS}$ or $\text{head} \rightarrow \text{body}$, where head contains only one non-terminal and body contains a collection of terminals and non-terminals..

Representative Grammars

- Example

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

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

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

$V = \{E, T, F\}$

$T = \{+, -, *, /, (,), \text{id}\}$

$S = \{E\}$

P :

$E \rightarrow E + T \quad T \rightarrow T / F$

$E \rightarrow E - T \quad T \rightarrow F$

$E \rightarrow T \quad F \rightarrow (E)$

$T \rightarrow T * F \quad F \rightarrow \text{id}$

Error handling

- Common programming errors
 - Lexical errors
 - Syntactic errors
 - Semantic errors
 - Lexical errors
- Error handler goals
 - Report the presence of errors clearly and accurately
 - Recover from each error quickly enough to detect subsequent errors
 - Add minimal overhead to the processing of correct programs

Error Recovery Strategies

- Error recovery strategies are used by the parser to recover from errors once it is detected.
- The simplest recovery strategy is to quit parsing with an error message for the first error itself

Panic Mode Recovery

- Panic mode error recovery is the easiest method of error-recovering strategy which prevents the parser from developing infinite loops.
- When parser finds an error in the statement, it ignores the rest of the statement by not processing the input.
- The parser intends to find designated set of synchronizing tokens by discarding input symbols one at a time.
- Synchronizing tokens may be delimiters, semicolon or } whose role in source program is clear.
- Advantages
 - Simplicity. Never get into infinite loop.
- Disadvantage
 - Additional errors cannot be checked as some of the input symbols will be skipped.

Phrase Level Recovery

- Parser performs local correction on the remaining input when an error is detected.
 - When a parser finds an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead.
 - One wrong correction will lead to an infinite loop.
 - The local correction may be
 - Replacing a prefix by some string.
 - Replacing comma by semicolon.
 - Deleting extraneous semicolon.
 - Insert missing semicolon.

Phrase Level Recovery

- Advantage
 - It can correct any input string.
- Disadvantage
 - It is difficult to cope up with actual error if it has occurred before the point of detection.

Error Production

- Productions which generate erroneous constructs are augmented to the grammar by considering common errors that occur.
- These productions detect the anticipated errors during parsing.
- Error diagnostics about the erroneous constructs are generated by the parser.

Global Correction

- There are algorithms which make changes to modify an incorrect string into a correct string.
- These algorithms perform minimal sequence of changes to obtain globally least-cost correction.
- When a grammar G and an incorrect string p is given, these algorithms find a parse tree for a string q related to p with smaller number of transformations.
- The transformations may be insertions, deletions and change of tokens.
- Advantage
 - It has been used for phrase level recovery to find optimal replacement strings.
- Disadvantage
 - This strategy is too costly to implement in terms of time and space.

Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
 - $E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$
 - Derivations for $-(id+id)$
 - $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$
 - $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

Leftmost Derivation

- At each and every step the leftmost non-terminal is expanded by substituting its corresponding production to derive a string.

$$E \rightarrow E + E \mid E * E \mid id$$

Let

$$w = id + id * id$$

$$\underset{lm}{E} \rightarrow E + E$$

$$\underset{lm}{E} \rightarrow id + E \quad [E \rightarrow id]$$

$$\underset{lm}{E} \rightarrow id + E * E \quad [E \rightarrow E * E]$$

$$\underset{lm}{E} \rightarrow id + id * E \quad [E \rightarrow id]$$

$$\underset{lm}{E} \rightarrow id + id * id \quad [E \rightarrow id]$$

Leftmost Derivation

- $S \rightarrow SS + | SS * | a$

{Students use leftmost derivations to derive the string $w=aa+a^*$ using the above production }

Solution:

Let

$$w = aa + a^*$$

$$\begin{array}{l} S \rightarrow SS* \\ \text{lm} \end{array}$$

$$\begin{array}{l} S \rightarrow SS + S * \\ \text{lm} \end{array} \quad [S \rightarrow SS +]$$

$$\begin{array}{l} S \rightarrow aS + S * \\ \text{lm} \end{array} \quad [S \rightarrow a]$$

$$\begin{array}{l} S \rightarrow aa + S * \\ \text{lm} \end{array} \quad [S \rightarrow a]$$

$$\begin{array}{l} S \rightarrow aa + a * \\ \text{lm} \end{array} \quad [S \rightarrow a]$$

Rightmost Derivation

- at each and every step the rightmost non-terminal is expanded by substituting its corresponding production to derive a string.

$$E \rightarrow E + E \mid E * E \mid id$$

Solution:

Let

$$w = id + id * id$$

$$\begin{array}{l} E \rightarrow E + E \\ \text{rm} \end{array}$$

$$\begin{array}{l} E \rightarrow E + E * E \\ \text{rm} \end{array} \quad [E \rightarrow E * E]$$

$$\begin{array}{l} E \rightarrow E + E * id \\ \text{rm} \end{array} \quad [E \rightarrow id]$$

$$\begin{array}{l} E \rightarrow E + id * id \\ \text{rm} \end{array} \quad [E \rightarrow id]$$

$$E \rightarrow id + id * id \quad [E \rightarrow id]$$

Rightmost Derivation

- $S \rightarrow SS + | SS * | a$

{*Students use rightmost derivations to derive the string $w=aa+a^*$ using the above productions}*}

Solution:

Let

$$w = aa + a^*$$

$$\begin{array}{l} S \rightarrow SS* \\ \text{rm} \end{array}$$

$$\begin{array}{l} S \rightarrow Sa * \\ \text{rm} \end{array} \quad [S \rightarrow a]$$

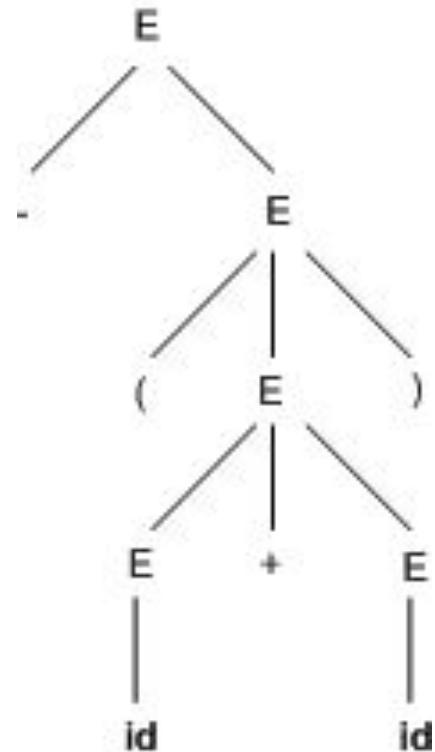
$$\begin{array}{l} S \rightarrow SS + a * \\ \text{rm} \end{array} \quad [S \rightarrow SS +]$$

$$\begin{array}{l} S \rightarrow Sa + a * \\ \text{rm} \end{array} \quad [S \rightarrow a]$$

$$\begin{array}{l} S \rightarrow aa + a * \\ \text{rm} \end{array} \quad [S \rightarrow a]$$

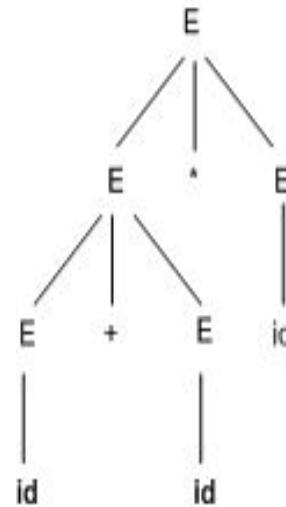
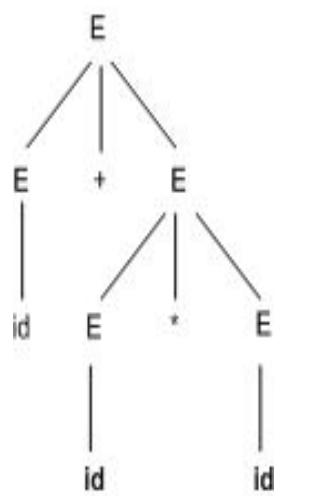
Parse tree

- Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings.
- Root node of parse tree has the start symbol of the given grammar from where the derivation proceeds.
- Leaves of parse tree represent terminals.
- Each interior node represents productions of grammar.
- If $A \rightarrow xyz$ is a production, then the parse tree will have A as interior node whose children are x , y and z from its left to right.



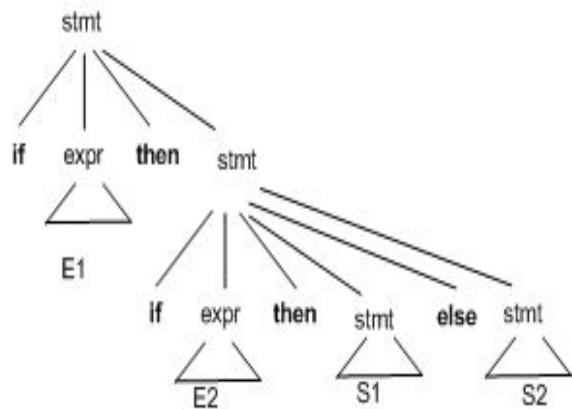
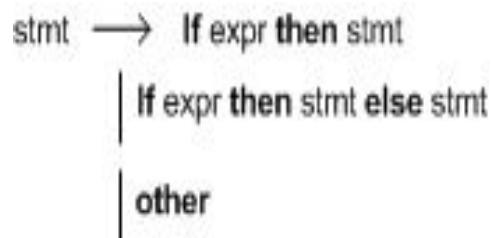
Ambiguity

- For some strings there exist more than one parse tree
- Or more than one leftmost derivation
- Or more than one rightmost derivation
- Example: $\text{id} + \text{id} * \text{id}$

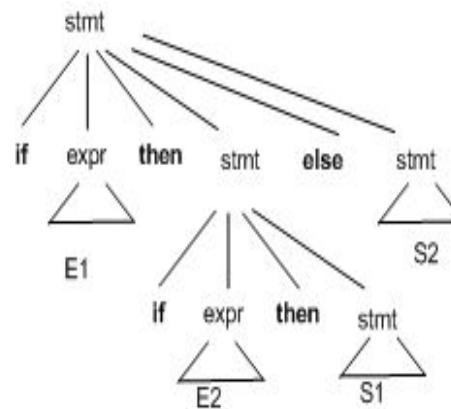
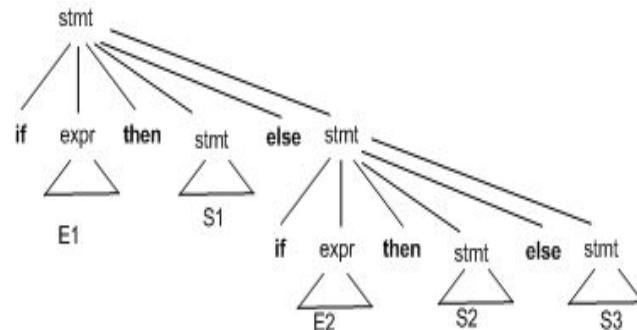


Elimination of ambiguity

if E1 then if E2 then S1 else S2

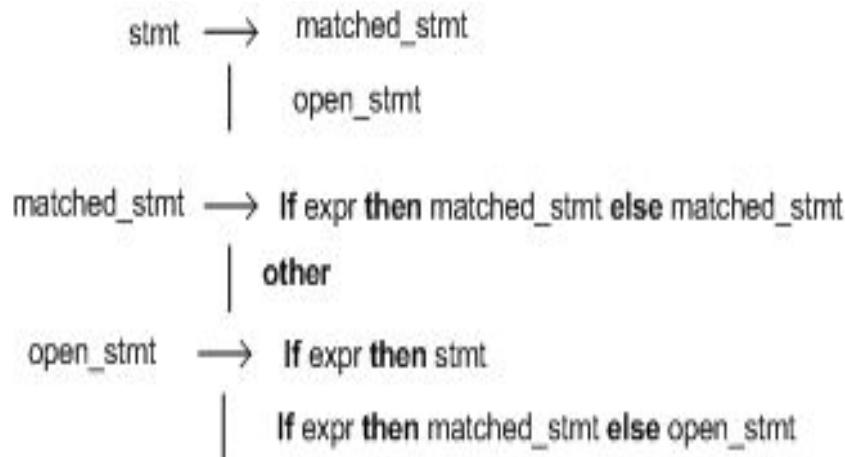


if E1 then S1 else if E2 then S2 else S3



Elimination of ambiguity (cont.)

- Idea:
 - A statement appearing between a **then** and an **else** must be matched



Eliminating left-recursion

- A grammar is left recursive if it has a production of the form $A \rightarrow A\alpha$, for some string α . To eliminate left-recursion for the production, $A \rightarrow A\alpha | \beta$
- Rule

$$\begin{array}{l} S \rightarrow Aa | b \\ A \rightarrow \beta A' \\ A' \rightarrow \alpha A' | \varepsilon \end{array}$$

$$\begin{array}{l} S \rightarrow Aa | b \\ A \rightarrow Ac | Sd | \varepsilon \end{array}$$

Solution:

Substitute the productions of S in A ,

$$A \rightarrow Ac | Aad | bd | \varepsilon$$

After eliminating left-recursion,

$$\begin{array}{l} S \rightarrow Aa | b \\ A \rightarrow bdA' | A' \\ A' \rightarrow cA' | adA' | \varepsilon \end{array}$$

Left Recursion Elimination Example

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

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow (E) \mid \mathbf{id}$$

Left factoring

- When a production has more than one alternatives with common prefixes, then it is necessary to make right choice on production.
- To perform left-factoring for the production, $A \rightarrow \alpha\beta_1|\alpha\beta_2$
- Rule

$$A \rightarrow \alpha A'$$

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

$$S \rightarrow iEtS | iEtSeS | a$$

$$E \rightarrow b$$

Solution:

$$S \rightarrow iEtSS' | a$$

$$S' \rightarrow eS | \epsilon$$

$$E \rightarrow b$$

Top-Down Parsing

- Top-down parsing constructs parse tree for the input string, starting from root node and creating the nodes of parse tree in pre-order.
- Top-down parsing is characterized by the following methods:
- ***Brute-force method***, accompanied by a parsing algorithm. All possible combinations are attempted before the failure to parse is recognized.
- ***Recursive descent***, is a parsing technique which does not allow backup. Involves backtracking and left-recursion.
- ***Top-down parsing*** with limited or partial backup.

Recursive Descent Parser

- Recursive descent parser is a top-down parser.
- It requires backtracking to find the correct production to be applied.
- The parsing program consists of a set of procedures, one for each non-terminal.
- Process begins with the procedure for start symbol.
- Start symbol is placed at the root node and on encountering each non-terminal, the procedure concerned is called to expand the non-terminal with its corresponding production.
- Procedure is called recursively until all non-terminals are expanded.
- Successful completion occurs when the scan over entire input string is done, i.e., all terminals in the sentence are derived by parse tree.

Recursive Descent Parser

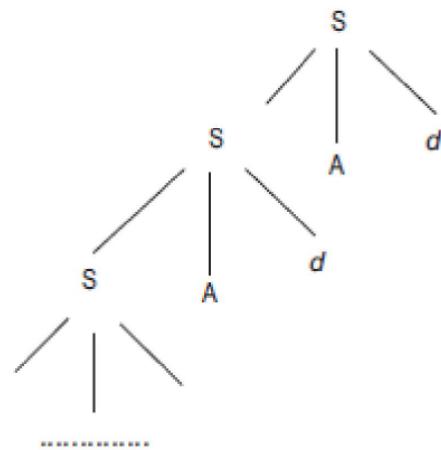
- **Limitation**

When a grammar with left recursive production is given, then the parser might get into infinite loop.

e.g., Let grammar G be,

$$S \rightarrow SAd$$

$$A \rightarrow ab \mid d$$

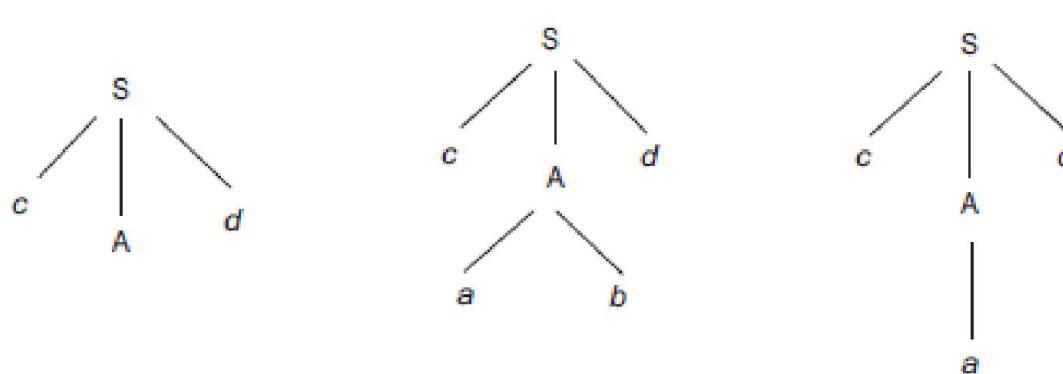


Recursive descent parser with backtracking

e.g., Let grammar G be,

$$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow ab \quad | \quad a \end{aligned}$$

For $w = cad$



Recursive descent parser with backtracking

- The root node contains the start symbol which is S .
- The body of production begins with c , which matches with the first symbol of the input string.
- A is a non-terminal which is having two productions $A \rightarrow ab \mid d$.
- Apply the first production of A , which results in the string $cabd$ that does not match with the given string cad .
- Backtrack to the previous step where the production of A gets expanded and try with alternate production of it.
- This produces the string cad that matches with the given string.

Recursive descent parser with backtracking

- Limitation:
 - If the given grammar has more number of alternatives then the cost of backtracking will be high

Recursive descent parser without backtracking

- Recursive descent parser without backtracking works in a similar way as that of recursive descent parser with backtracking with the difference that each non-terminal should be expanded by its correct alternative in the first selection itself.
- When the correct alternative is not chosen, the parser cannot backtrack and results in syntactic error

Advantage

–Overhead associated with backtracking is eliminated.

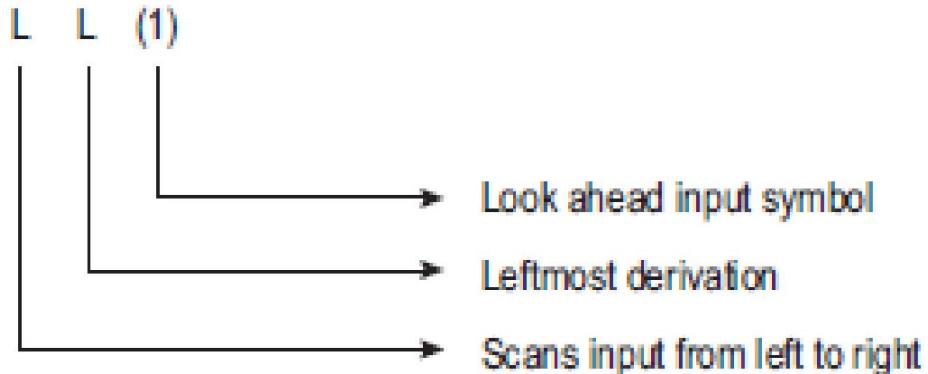
Limitation

-- When more than one alternative with common prefixes occur, then the selection of the correct alternative is highly difficult.

Predictive Parser / LL(1) Parser

- Predictive parsers are top-down parsers.
- It is a type of recursive descent parser but with no backtracking.
- It can be implemented non-recursively by using stack data structure.
- They can also be termed as LL(1) parser as it is constructed for a class of grammars called LL(1).
- The production to be applied for a non-terminal is decided based on the current input symbol.

Predictive Parser / LL(1) Parser



- In order to overcome the limitations of recursive descent parser, LL(1) parser is designed by using stack data structure explicitly to hold grammar symbols.
- In addition to this, Left-recursion is eliminated.
- Common prefixes are also eliminated (left-factoring).

Computation of FIRST

- $\text{FIRST}(\alpha)$ is the set of terminals that begin strings derived from α .

Rules

- To compute $\text{FIRST}(X)$, where X is a grammar symbol,
 - If X is a terminal, then $\text{FIRST}(X) = \{X\}$.
 - If $X \rightarrow \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.
 - If X is a non-terminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then add $\text{FIRST}(Y_1)$ to $\text{FIRST}(X)$. If Y_1 derives ϵ , then add $\text{FIRST}(Y_2)$ to $\text{FIRST}(X)$.

Computation of FOLLOW

- FOLLOW(A) is the set of terminals a, that appear immediately to the right of A. For rightmost sentential form of A, \$ will be in FOLLOW(A).
- Rules
 - For the FOLLOW(start symbol) place \$, where \$ is the input end marker.
 - If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
 - If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ , then everything in FOLLOW(A) is in FOLLOW(B).

Construction of parsing table

Algorithm Construction of predictive parsing table

Input Grammar G

Output Parsing table M

Method For each production $A \rightarrow \alpha$, do the following:

1. For each terminal a in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $\text{FIRST}(\alpha)$, then for each terminal b in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$.
3. If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$.
4. If no production is found in $M[A, a]$ then set error to $M[A, a]$.

Parsing of input

- Predictive parser contains the following components:
 - Stack – holds sequence of grammar symbols with \$ on the bottom of stack
 - Input buffer – contains the input to be parsed with \$ as an end marker for the string.
 - Parsing table.

Parsing of input - Process

- Initially the stack contains \$ to indicate bottom of the stack and the start symbol of grammar on top of \$.
- The input string is placed in input buffer with \$ at the end to indicate the end of the string.
- Parsing algorithm refers the grammar symbol on the top of stack and input symbol pointed by the pointer and consults the entry in $M[A, a]$ where A is in top of stack and a is the symbol read by the pointer.
- Based on the table entry, if a production is found then the tail of the production is pushed onto stack in reversal order with leftmost symbol on the top of stack.
- Process repeats until the entire string is processed.

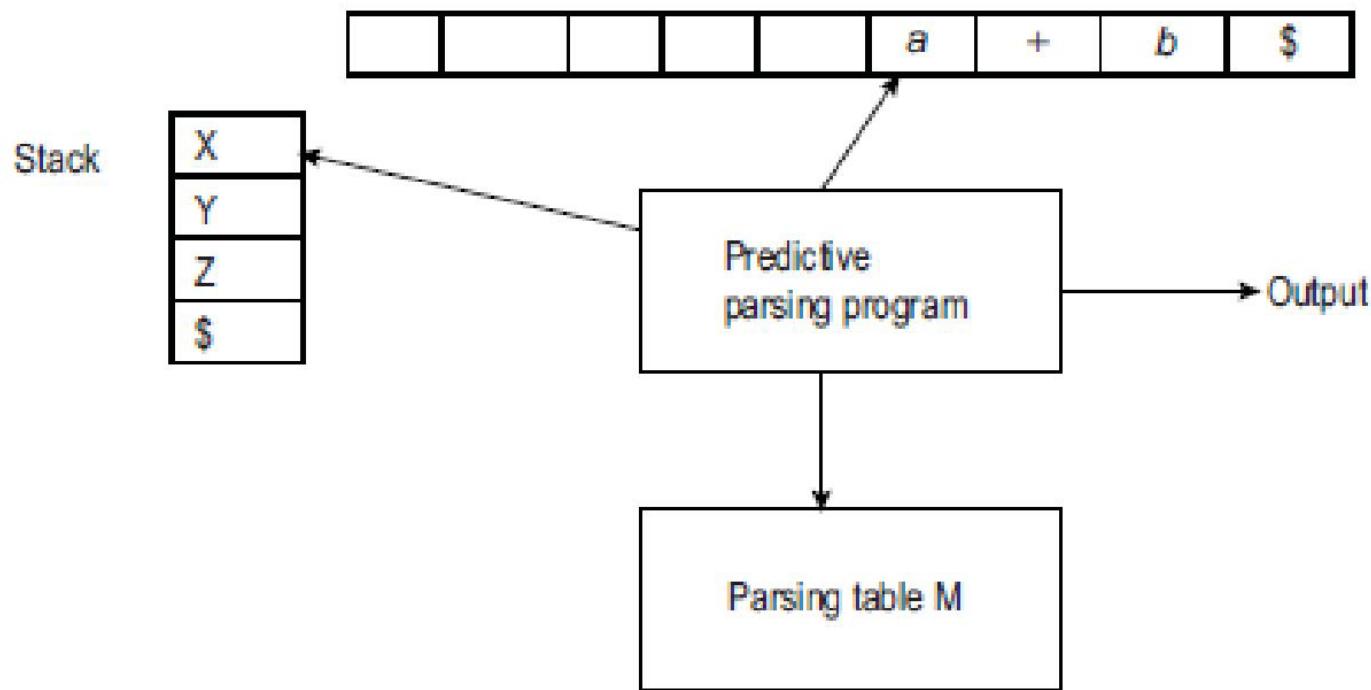
Parsing of input - Process

- When the stack contains \$ (bottom end marker) and the pointer reads \$ (end of input string), successful parsing occurs.
- If no entry is found, it reports error stating that the input string cannot be parsed by the grammar

Non-recursive Predictive Parser

- Non-recursive predictive parser uses explicit stack data structure.
- This prevents implicit recursive calls.
- It can also be termed as table-driven predictive parser.
- Components
 - Input buffer – holds input string to be parsed.
 - Stack – holds sequence of grammar symbols.
 - Predictive parsing algorithm – contains steps to parse the input string; controls the parser's process.
 - Parsing table – contains entries based on which parsing actions has to be carried out.

Model of a table-driven predictive parser



Process

- Initially, the stack contains \$ at the bottom of the stack.
- The input string to be parsed is placed in the input buffer with \$ as the end marker.
- If X is a non-terminal on the top of stack and the input symbol being read is a, the parser chooses a production by consulting entry in the parsing table M[X, a].

Process

- Replace the non-terminal in stack with the production found in $M[X, a]$ in such a way that the leftmost symbol of right side of production is on the top of stack, i.e., the production has to be pushed to stack in reverse order.
- Compare the top of stack symbol with input symbol.
- If it matches, pop the symbol from stack and advance the pointer reading the input buffer.
- If no match is found repeat from step 2. Stop parsing when the stack is empty (holds $\$$) and input buffer reads end marker ($\$$).

Example

- Construct predictive parsing table for the grammar,

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

and parse the input $id + id * id$.

Solution

- Step 1: Eliminate left-recursion

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

$$V = \{E, T, F, E', T'\}$$

$$T = \{+, *, (,), \text{id}\}$$

Solution

- Step 2: Left-factoring No common prefixes for any production with same head, i.e., no need of left-factoring

Step 3: Compute FIRST

$\text{First}(\text{terminal}) = \{\text{terminal}\}$

$$\text{FIRST}(+) = (+) \quad \text{FIRST}(*) = (*)$$

$$\text{FIRST}(()) = \{()\} \quad \text{FIRST}(()) = \{()\}$$

$$\text{FIRST}(\text{id}) = \{\text{id}\}$$

$$\text{FIRST}(E) = \{(), \text{id}\}$$

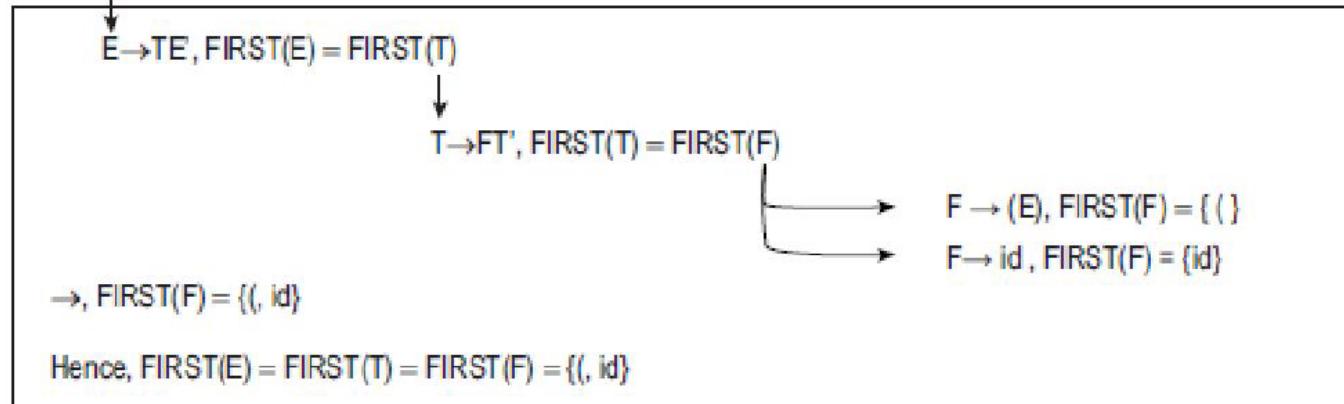


Figure 2.10 Computing FIRST for E, T and F

$$\text{FIRST}(T) = \{(), \text{id}\} \quad [\text{Refer Figure 3.10}]$$

$$\text{FIRST}(F) = \{(), \text{id}\} \quad [\text{Refer Figure 3.10}]$$

Step 3: Compute FIRST

If a non-terminal derives ϵ , then ϵ is in FIRST (non-terminal).

i.e., if $A \rightarrow \epsilon$, then $\text{FIRST}(A) = \{\epsilon\}$

$$\text{FIRST}(E') = \{+, \epsilon\}$$

[Since, $E' \rightarrow + TE'$ is a production starts with a terminal '+', $\text{FIRST}(E')$ includes '+']

$$\text{FIRST}(T') = \{*, \epsilon\}$$

[Since, $T' \rightarrow *FT'$ is a production starts with a terminal '*', $\text{FIRST}(T')$ includes '*']

Step 4: Compute FOLLOW

Rule Place \$ in FOLLOW (start symbol)
 $\text{FOLLOW}(E) = \{\$, \}\}$

[Since, $F \rightarrow (E)$ is a production where E is followed by a terminal ') ', $\text{FOLLOW}(E)$ contains ') ']

Since the production, $F \rightarrow (E)$
 $\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$
 A α B β

i.e., $\text{FOLLOW}(E) = \text{FIRST}(\beta) = \{\}\}$

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

Since the production, $E \rightarrow T \ E'$
 $\downarrow \quad \downarrow \quad \downarrow$
 A α B

i.e., $\text{FOLLOW}(E') = \text{FOLLOW}(E) = \{\), \$\}$

Step 4: Compute FOLLOW

$$\text{FOLLOW}(T) = \{+,), \$\}$$

Since the production, $E' \rightarrow + T E'$
 $\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$
 $A \quad \alpha \quad B \quad \beta$

$$\text{i.e., } \text{FOLLOW}(T) = \text{FIRST}(\beta) = \text{FIRST}(E') = \{+, \varepsilon\}$$

Since ε is in $\text{FIRST}(E')$, everything in $\text{FOLLOW}(E')$ is in $\text{FOLLOW}(T)$.

$$\text{i.e., } \text{FOLLOW}(B) = \text{FOLLOW}(A) \cup \{\text{FIRST}(\beta) - \varepsilon\}$$

$$\begin{aligned}\text{FOLLOW}(T) &= \text{FOLLOW}(E') \cup \{\text{FIRST}(E') - \varepsilon\} \\ &= \{\$,)\} \cup [\{+, \varepsilon\} - \varepsilon] \\ &= \{\$,), +\}\end{aligned}$$

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

Since the production, $T \rightarrow F T'$
 $\downarrow \quad \downarrow \quad \downarrow$
 $A \quad \alpha \quad B$

$$\text{i.e., } \text{FOLLOW}(B) = \text{FOLLOW}(A)$$

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

Step 4: Compute FOLLOW

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

Since the production, $T' \rightarrow * F T'$
 $\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$
 $A \quad \alpha \quad B \quad \beta$

$$\text{i.e., } \text{FOLLOW}(F) = \text{FIRST}(\beta) = \text{FIRST}(T') = \{*, \epsilon\}$$

Since ϵ is in $\text{FIRST}(T')$, everything in $\text{FOLLOW}(T')$ is in $\text{FOLLOW}(F)$.

$$\text{i.e., } \text{FOLLOW}(B) = \text{FOLLOW}(A) \cup \{\text{FIRST}(\beta) - \epsilon\}$$

$$\begin{aligned}\text{FOLLOW}(F) &= \text{FOLLOW}(T') \cup \{\text{FIRST}(T') - \epsilon\} \\ &= \{\$,), +\} \cup [\{*, \epsilon\} - \epsilon] \\ &= \{\$,), +, *\}\end{aligned}$$

Step 5: Construct parsing 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	$F \rightarrow id$			$F \rightarrow (E)$		

Step 6: Parse the given input

$w = \text{id} + \text{id}^* \text{id}$

Stack	Input	Action
E \$	id + id * id \$	
T E' \$	id + id * id \$	Replace E by its production $E \rightarrow TE'$ in M[E, id]
FT' E' \$	id + id * id \$	Replace T by its production $T \rightarrow FT'$ in M[T, id]
id T' E' \$	id + id * id \$	Replace F by its production $F \rightarrow \text{id}$ in M[F, id]
id T' E' \$	id + id * id \$	Match id
T' E' \$	+ id * id \$	
ϵ E' \$	+ id * id \$	Replace T' by its production $T' \rightarrow \epsilon$ in M[T', +]
E' \$	+ id * id \$	
+ TE' \$	+ id * id \$	Replace E' by its production $E' \rightarrow + TE'$ in M[E', +]
+ TE' \$	+ id * id \$	Match +

Stack	Input	Action
$TE' \$$	$id * id \$$	
$FT' E' \$$	$id * id \$$	Replace T by its production $T \rightarrow FT'$ in $M[T, id]$
$id T' E' \$$	$id * id \$$	Replace F by its production $F \rightarrow id$ in $M[F, id]$
$id T' E' \$$	$id * id \$$	Match id
$T' E' \$$	$* id \$$	
$* F T' E' \$$	$* id \$$	Replace T' by its production $T' \rightarrow *FT'$ in $M[T', *]$
$* F T E' \$$	$* id \$$	Match *
$FT' E' \$$	$id \$$	
$id T' E' \$$	$id \$$	Replace F by its production $F \rightarrow id$ in $M[F, id]$
$id T' E' \$$	$id \$$	Match id
$T' E' \$$	$\$$	
$\epsilon E' \$$	$\$$	Replace F by its production $T \rightarrow \epsilon$ in $M[T, \$]$
$E' \$$	$\$$	
$\epsilon \$$	$\$$	Replace E' by its production $E' \rightarrow \epsilon$ in $M[E', \$]$
$\$$	$\$$	Successful parsing, i.e., accept the string

Example 2

- Construct predictive parsing table for the grammar,

$$S \rightarrow S(S)S \mid \epsilon$$

with the input (() ()).