Compiler Design

Unit II

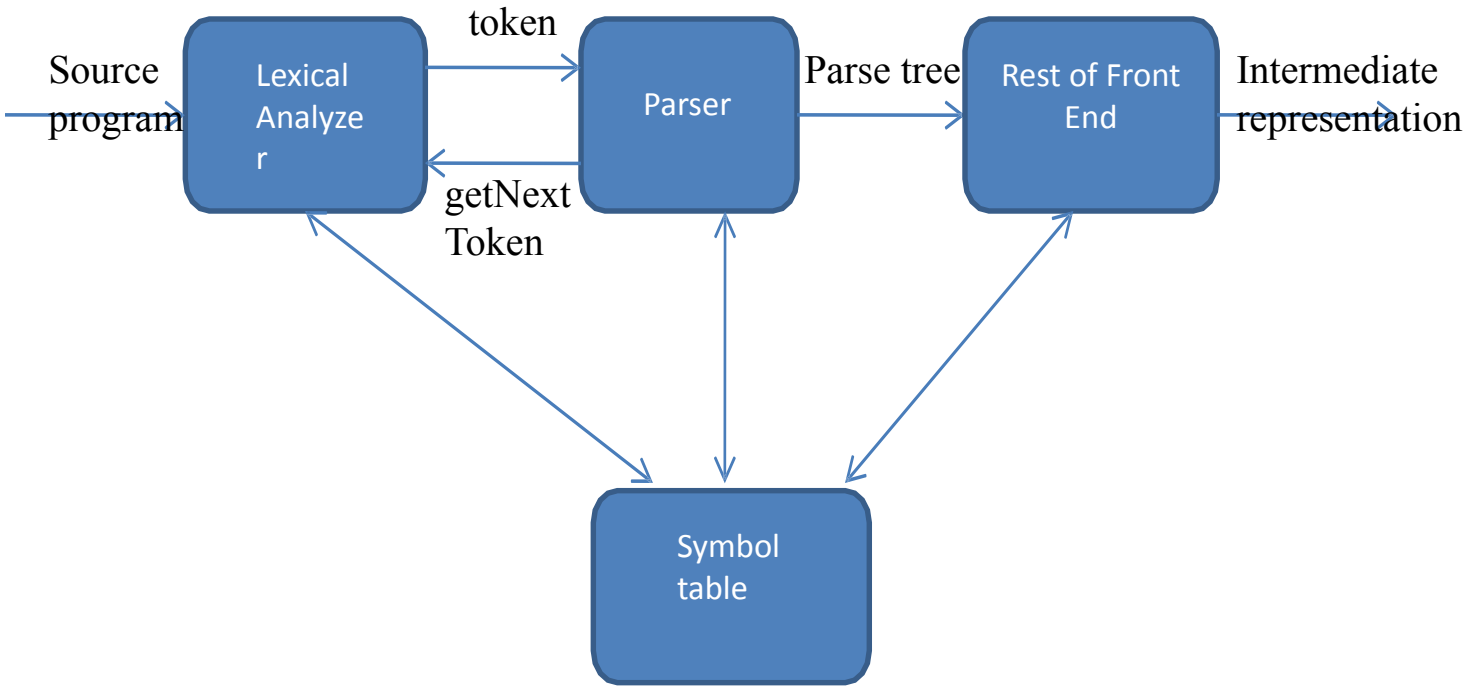
Syntax Analysis

- 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 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, paranthesis.
  Digits, i.e., 0, 1, 2, $\cdots$,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 LHS $\rightarrow$ RHS or $head \rightarrow body$, where head contains only onenon-terminal and body contains a collection of terminals and non-terminals..

Representative Grammars

- Example

E → E + T | E - T | T
T → T * F | T / F | F
F → (E) | **id**

V = {E, T, F}
T = {+ , -, *, /, (,), **id**}
S = {E}
P :

E → E+T	T→ T / F
E → E-T	T→ F
E →T	F→ (E)
T → T*F	F→ **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.

Derivation is a sequence of production rules.
• It is used to get the input string through these production rules.

Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations

– E -> E + E | E * E | -E | (E) | **id**

  – Derivations for **–(id+id)**

    - E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)
    - E => -E => -(E) => -(E+E) => -(E+**id**)=>-(**id**+**id**)

Leftmost Derivation

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

$E \rightarrow E + E \mid E * E \mid \mathbf{id}$

Let

$$w = \mathbf{id} + \mathbf{id} * \mathbf{id}$$

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

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

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

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

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

We have to decide the non-terminal which is to be replaced.
• We have to decide the production rule by which the non-terminal will be
replaced.

In the leftmost derivation, the input is scanned and replaced with the production rule from left to
right. So in leftmost derivation, we read the input string from left to right.

In rightmost derivation, the input is scanned and replaced with the production rule
from right to left. So in rightmost derivation, we read the input string from right to lef

Leftmost Derivation

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

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

Solution:

Let

$$w = aa + a*$$

$$S \underset{lm}{\rightarrow} SS*$$

$$S \underset{lm}{\rightarrow} SS + S * \qquad [S \rightarrow SS+]$$

$$S \underset{lm}{\rightarrow} aS + S * \qquad [S \rightarrow a]$$

$$S \underset{lm}{\rightarrow} aa + S * \qquad [S \rightarrow a]$$

$$S \underset{lm}{\rightarrow} aa + a * \qquad [S \rightarrow a]$$

Rightmost Derivation

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

  $E \rightarrow E + E \mid E * E \mid \mathbf{id}$

Solution:

Let

$$w = \mathbf{id} + \mathbf{id} * \mathbf{id}$$

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

$$E \underset{rm}{\rightarrow} E + E * E \qquad [E \rightarrow E * E]$$

$$E \underset{rm}{\rightarrow} E + E * \mathbf{id} \qquad [E \rightarrow \mathbf{id}]$$

$$E \underset{rm}{\rightarrow} E + \mathbf{id} * \mathbf{id} \qquad [E \rightarrow \mathbf{id}]$$

$$E \rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \qquad [E \rightarrow \mathbf{id}]$$

Rightmost Derivation

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

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

Solution:

Let

$$w = aa + a*$$

$$S \underset{rm}{\rightarrow} SS*$$

$$S \underset{rm}{\rightarrow} Sa* \qquad [S \rightarrow a]$$

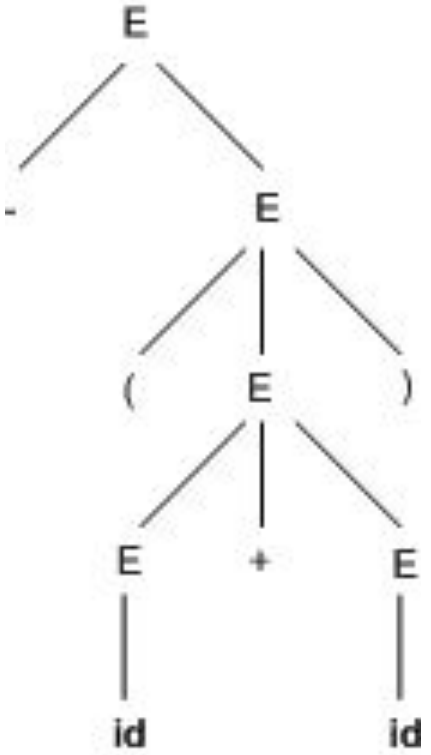
$$S \underset{rm}{\rightarrow} SS+a* \qquad [S \rightarrow SS+]$$

$$S \underset{rm}{\rightarrow} Sa+a* \qquad [S \rightarrow a]$$

$$S \underset{rm}{\rightarrow} aa+a* \qquad [S \rightarrow a]$$

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.

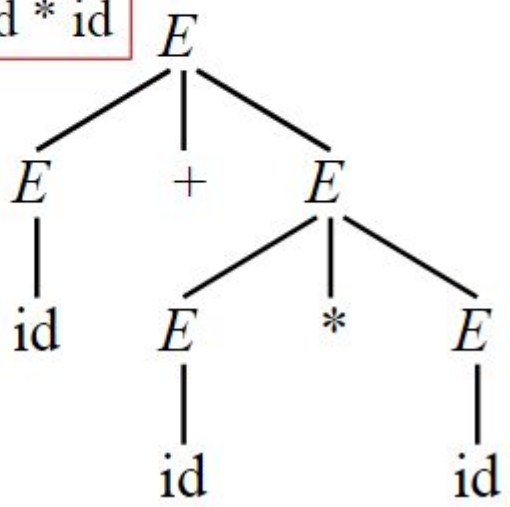


Ambiguity

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string. If the grammar is not ambiguous then it is called unambiguous.

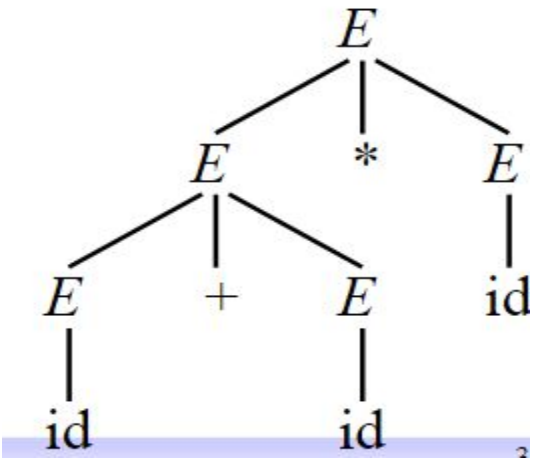
LMD

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



RMD

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



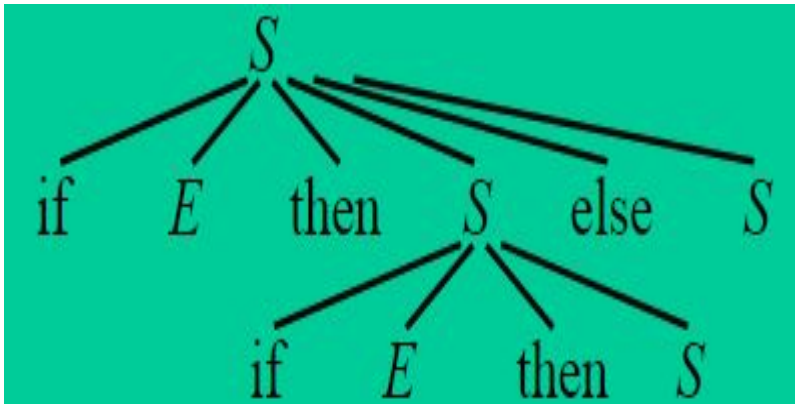
Elimination of ambiguity

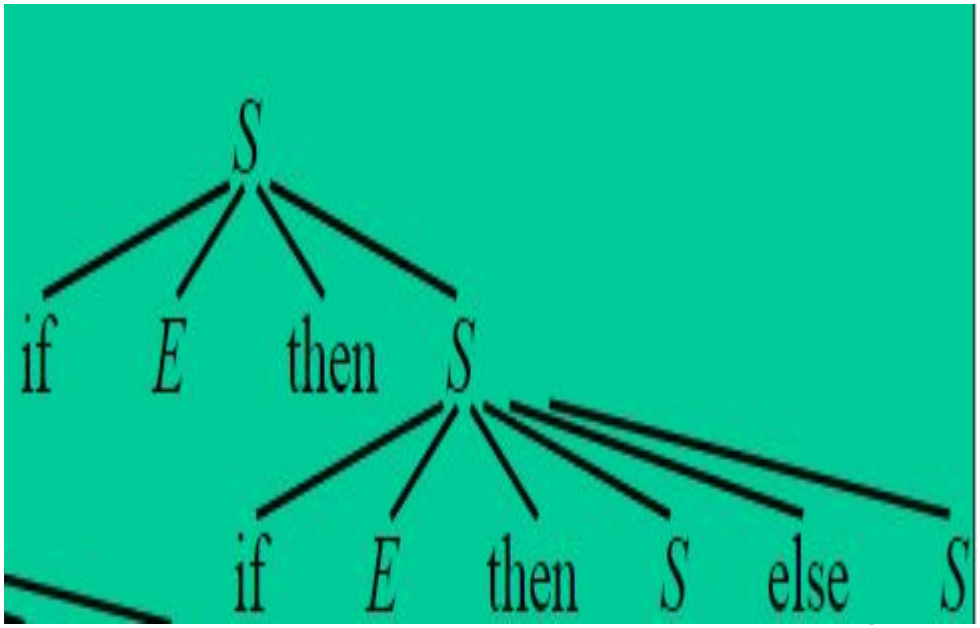
Example

The dangling-else grammar

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

if E_1 then if E_2 then S_1 else S_2





Elimination of ambiguity (cont.)

Disambiguating Rules

* Rule: match each else with the closest previous unmatched then
* Remove undesired state transitions in the pushdown automaton

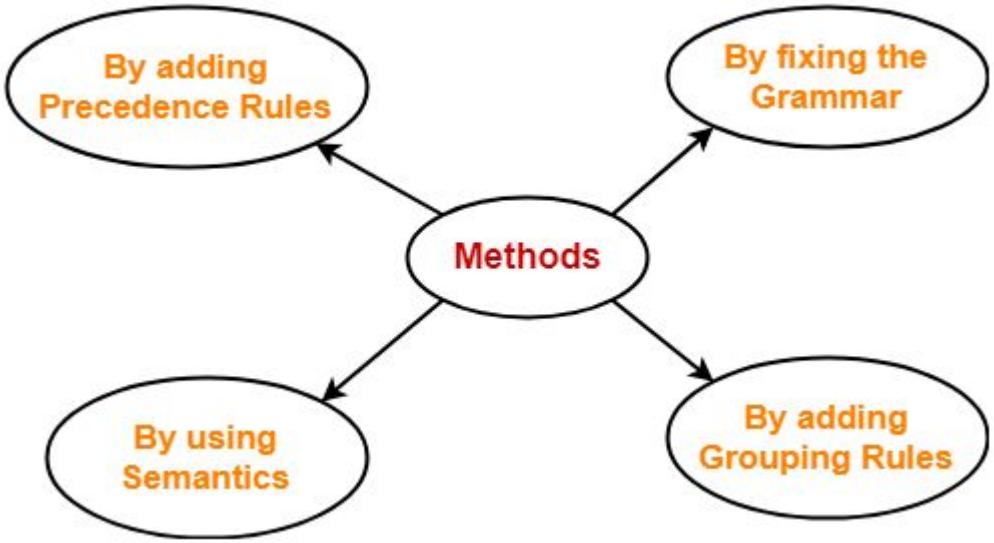
Grammar Rewriting

stmt → *m_stmt*
| *unm_stmt*

m_stmt → **if** *expr* **then** *m_stmt* **else** *m_stmt*
| **other**

unm_stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *m_stmt* **else** *unm_stmt*

Elimination of ambiguity (cont.)



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, $A \rightarrow A\ \alpha \mid \beta$.

- If a left recursion is present in any grammar then, during parsing in the the syntax analysis part of compilation there is a chance that the grammar will create infinite loop. This is because at every time of production of grammar S will produce another S without checking any condition.

- Left recursive grammar: $A \rightarrow A\ \alpha \mid \beta$
- After elimination of LR:

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

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, $A \rightarrow A\ \alpha \mid \beta$.

- If a left recursion is present in any grammar then, during parsing in the the syntax analysis part of compilation there is a chance that the grammar will create infinite loop. This is because at every time of production of grammar S will produce another S without checking any condition.

- Left recursive grammar: $A \rightarrow A\ \alpha \mid \beta$
- After elimination of LR:

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

Left Recursion Elimination Example

E -> E+T | T

T -> T*F | F

F -> (E) | **id**

E -> TE’

E’ -> +TE’ | ε

T -> FT’

T’ -> *FT’ | ε

F -> (E) | **id**

1. S□Sad|ba
2. A□Aiu|Abu

Left factoring

- When a production has more than one alternatives with common prefixes, then it is necessary to make right choice on production.
- It is a grammar transformation technique.
- Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracing by the parser
- To perform left-factoring for the production, $A \rightarrow \alpha\beta 1 | \alpha\beta 2$
- Rule

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

$$S \rightarrow i\text{Et } S \mid i\text{Et } SeS \mid a$$
$$E \rightarrow b$$

Solution:

$$S \rightarrow i\text{Et}SS' \mid a$$
$$S' \rightarrow eS \mid \varepsilon$$
$$E \rightarrow b$$

Left factoring

1. Left recursion: when one or more productions can be reached from themselves with no tokens consumed in-between.

2. Left factoring: a process of transformation, turning the grammar from a left-recursive form to an equivalent non-left-recursive form.

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.

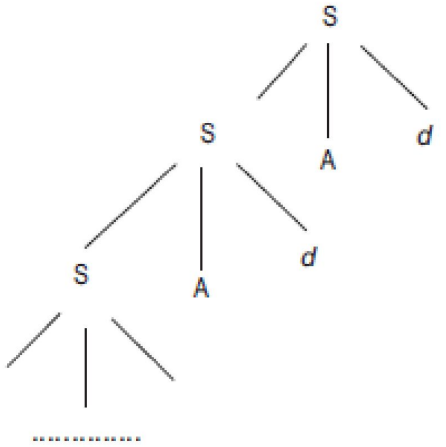
Recursive Descent Parser

- **Limitation**

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

e.g., Let grammar G be,

$$S \rightarrow SAd$$
$$A \rightarrow ab \mid d$$



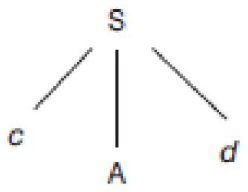
Recursive descent parser with backtracking

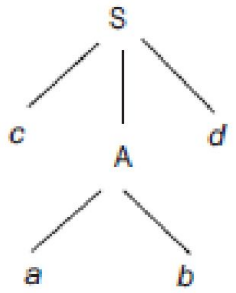
- The root node contains the start symbol which is S.
- The body of production begins with *c*, which matches with the first symbol of the input string.
- A is a non-terminal which is having two productions A ⟶ ab | d.
- Apply the first production of A, which results in the string ***cabd*** that does 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.

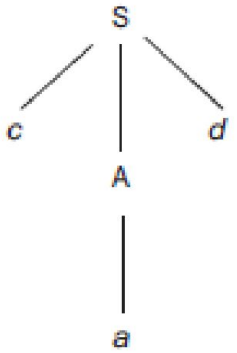
Recursive descent parser with backtracking

e.g., Let grammar G be,

S → cAd
A → ab | a
For w = cad







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.

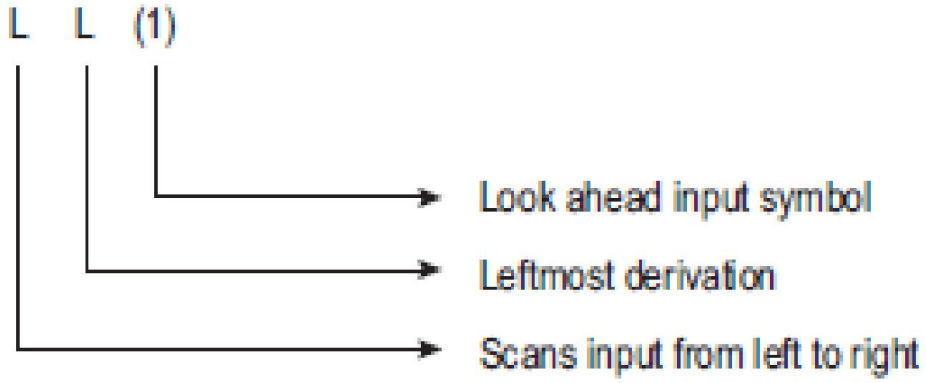
Limitation

-- 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).

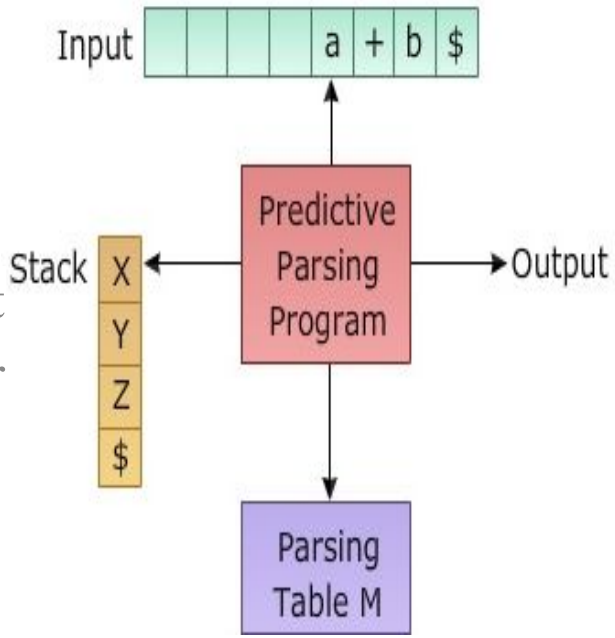
Predictive Parser / LL(1) Parser

INPUT: Contains string to be parsed with $ as it's end marker

STACK: Contains sequence of grammar symbols with $ as it's bottom marker. Initially stack contains only $

PARSING TABLE: A two dimensional array M[A,a], where A is a non-terminal and a is a Terminal

It is a tabular implementation of the recursive descent parsing, where a stack is maintained by the parser rather than the language in which parser is written



Implementation of a Predictive Parser:

Predictive parser use an algorithm to parse the input string, which makes use of entries in a table which are filled by using another algorithm using two important set computed for the given gramma

Steps in Predictive Parsing

- Eliminate Left Recursion
- Left factor the grammar
- Compute FIRST () and FOLLOW()
- Build the predictive parsing table using the FIRST() and FOLLOW() elements

Computation of FIRST

- FIRST(α) is the set of terminals that begin strings derived from α.

Rules

- To compute FIRST(X), where X is a grammar symbol,
  - If X is a terminal, then FIRST(X) = {X}.
  - If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
  - If X is a non-terminal and $X \rightarrow Y1\ Y2 \cdots Yk$ is a production, then add FIRST(Y1) to FIRST(X). If Y1 derives ε, then add FIRST(Y2) to FIRST(X).

1. $S \rightarrow A\ a$
2. $A \rightarrow B\ D$
3. $B \rightarrow b$
4. $B \rightarrow \epsilon$
5. $D \rightarrow d$
6. $D \rightarrow \epsilon$

1. FIRST(a)=a
   - **FIRST(b)=b**
   - **FIRST(d)=d**
   - **FIRST(S)=FIRST(A)={a,b,d}**
   - **FIRST(A)=FIRST(B)={b, d, ε}**
   - **FIRST(B)={ε,b }**
   - **FIRST(D)={ε, d}**

S□ACB|CbB|Ba
A□da|BC
B□g| ε
C□h|ε

FIRST(S)={a, b, d, g, h,ε}

FIRST(A)={d,g,h,ε}

FIRST(B)={g, ε}

FIRST(C)={h, ε}

2. S□aABb

A□c|ε

B□d|ε

3. S□ABCDE

A□a| ε

B□b| ε

C□c

D□d ε

E□e| ε

3. S→ABCDE
A→a| ε
B→b| ε
C→c
D→d ε
E→e| ε

FIRST(S)={a, b,c}
FIRST(A)={ a, ε}
FIRST(B)={b, ε
FIRST(C)={ c}
FIRST(D)={d, ε}
FIRST(E)={e,ε}

2. S→aABb
A→c|ε
B→d|ε

FIRST(S)=a
FIRST(A)={c, ε
FIRST(B)={d, ε

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

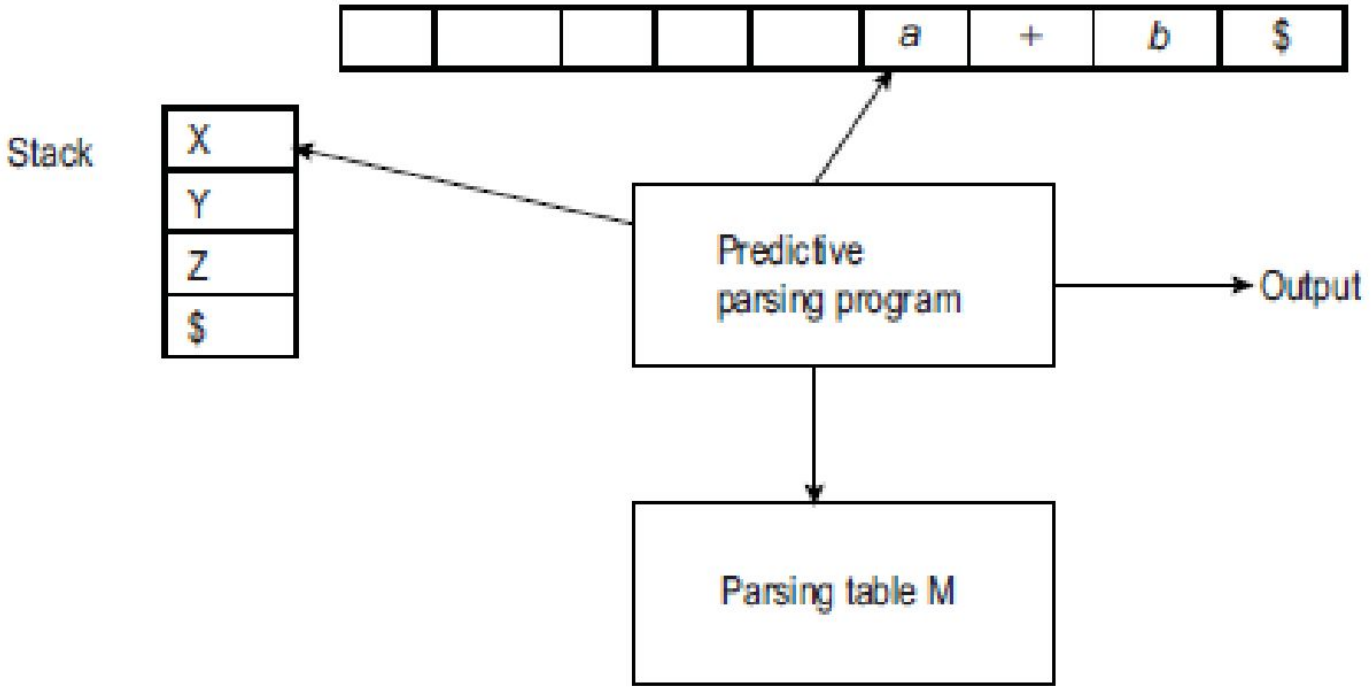
Parsing of input - Process

- 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

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

$$V = \{E, T, F, E', T'\}$$
$$T = \{+, *, (,), \mathbf{id}\}$$

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

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

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

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

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

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

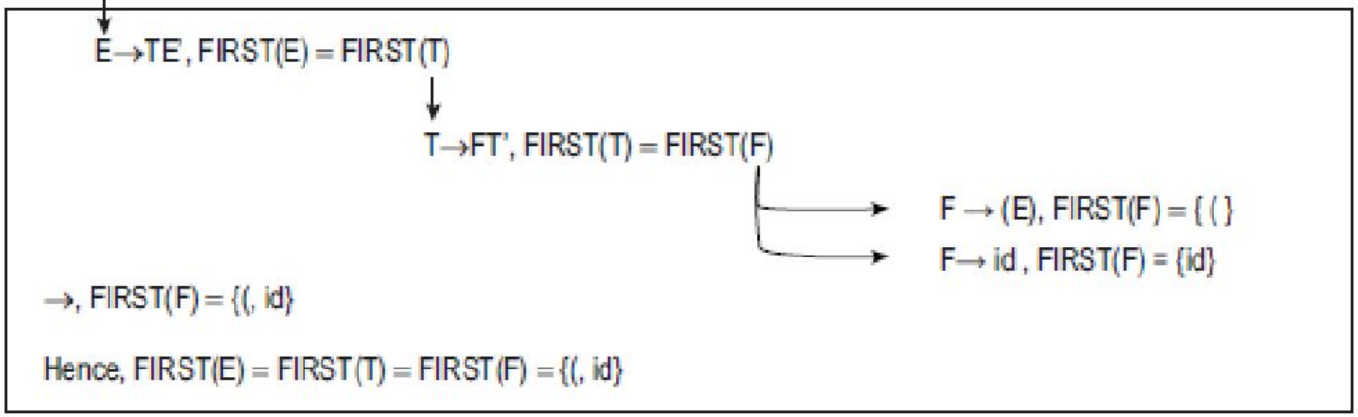


Figure 2.10 Computing FIRST for E, T and F

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

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

Step 3: Compute FIRST

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

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

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

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

Step 4: Compute FOLLOW

Rule Place \$ in FOLLOW (start symbol) FOLLOW (E) = {\$,)}	[Since, F → (E) is a production where E is followed by a terminal ′) ′, FOLLOW(E) contains ′) ′] Since the production, F → (E) ↓ ↓ ↓ ↓ A α B β i.e., FOLLOW(E) = FIRST(β) = {)}
FOLLOW(E′) = {\$,)}	Since the production, E → T E′ ↓ ↓ ↓ A α B i.e., FOLLOW(E′) = FOLLOW(E) = {), \$}

Step 4: Compute FOLLOW

$\text{FOLLOW(T)} = \{+,), \$\}$	Since the production, $\text{E}' \rightarrow + \text{ T } \text{E}'$ $\downarrow \quad \downarrow \downarrow \downarrow$ $\text{A} \quad \alpha \text{ B } \beta$ i.e., $\text{FOLLOW(T)} = \text{FIRST}(\beta) = \text{FIRST}(\text{E}') = \{+, \varepsilon\}$ Since ε is in $\text{FIRST}(\text{E}')$, everything in $\text{FOLLOW}(\text{E}')$ is in FOLLOW(T). i.e., $\text{FOLLOW(B)} = \text{FOLLOW(A)} \cup \{\text{FIRST}(\beta) - \varepsilon\}$ $\text{FOLLOW(T)} = \text{FOLLOW}(\text{E}') \cup \{\text{FIRST}(\text{E}') - \varepsilon\}$ $= \{\$,)\} \cup [\{+, \varepsilon\} - \varepsilon]$ $= \{\$,), +\}$
$\text{FOLLOW}(\text{T}') = \{+,), \$\}$	Since the production, $\text{T} \rightarrow \text{F } \text{T}'$ $\downarrow \quad \downarrow \downarrow$ $\text{A} \quad \alpha \text{ B}$ i.e., $\text{FOLLOW(B)} = \text{FOLLOW(A)}$ $\text{FOLLOW}(\text{T}') = \text{FOLLOW(T)} = \{+,), \$\}$

Step 4: Compute FOLLOW

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

Since the production,

$$\begin{array}{cccccc} \mathrm{T}' & \rightarrow & * & \mathrm{F} & \mathrm{T}' \\ \downarrow & & \downarrow & \downarrow & \downarrow \\ \mathrm{A} & & \alpha & \mathrm{B} & \beta \end{array}$$

i.e., $\mathbf{FOLLOW(F)} = \mathbf{FIRST}(\beta) = \mathbf{FIRST}(\mathrm{T}') = \{*, \varepsilon\}$

Since ε is in FIRST(T'), everything in **FOLLOW** (T') is in **FOLLOW(F)**.

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

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

Step 5: Construct parsing table

For each production A→α do

For each a in FIRST(α), add A→α to M[A, a]

If ϵ is in FIRST(α), add A→α to M[A, b]
for each token b in FOLLOW(A)
If ϵ is in FIRST(α) and $ is in FOLLOW(A) then
add A→α to M[A, $]

Mark all other entries of M as "error"

Non-terminal	Input symbol					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ id			$F \rightarrow (E)$		

Step 6: Parse the given input

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

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

Example 2

- Construct predictive parsing table for the grammar,

  $S \rightarrow S(S)S \mid \varepsilon$

  with the input (( ) ( )).