Unit II

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 *non terminals*

– 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
  $\alpha, \beta, \gamma \in (N \cup T)^*$|

Representative Grammars

- **The syntax of the programming language constructs can be described by context free grammars or BNF (Backnus-Naur Form).**
- 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.

Context Free Grammars and Languages

- Context free grammar (CFG) is a formal grammar which is used to generate all possible strings in a given formal language.
- Context free grammar G can be defined by four tuples as:

  (N, T, P, S)
- Where,
- N:- Set of Non terminals or variable list
- T:- Set of Terminals($T \in \sum$)
- S:- Special Non terminal called Starting symbol of grammar($S \in N$)
- P:- Production rule (of the form $\alpha \rightarrow \beta$, where α and β are strings on $N \cup \sum$)
- In CFG, the start symbol is used to derive the string.
- We can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.
- It is used to generate all possible patterns of strings in a given formal language.

Context Free Grammar:

- Type-2 grammars generate the context-free languages.
- The language generated by the grammar is recognized by a Pushdown automata (PDA)

Rules:

1. Left hand side of production can have only one variable.

Where,

α is Single NT

β is (V + T)*.

Example

S –> AB
A –> a/ε
B –> b

Examples

Example 1:

Construct the CFG for the language having any number of a's over the set ∑= {a}. R.E= a*

Grammar :Production rule (P):

S → aS rule 1

S → ε rule 2

Derive a string aaa

-> S

->aS

->aaS rule 1

->aaaS rule 1

->aaa ε rule 2

-> aaa (Required string)

Example 2:

Construct a CFG for the regular expression (0 +1)*

Grammar :Production rule (P):

S → 0S | 1S rule 1

S → ε rule 2

Derive a string “1001”

->S

->1S rule 1

->10S rule 1

-> 100S rule 1

-> 1001S rule 1

-> 1001ε rule 2

-> 1001 (Required string)

Example 3:

Construct a CFG for defining palindrome over ∑={a,b}, L = {wcw^R}

Grammar :Production rule (P):

S → aSa rule 1

S → bSb rule 2

S → c rule 3

Derive a string "abbcbba“

S → aSa

→ abSba from rule 2

→ abbSbba from rule 2

→ abbcbba from rule 3 (Required string)

Example 4:

Construct a CFG for defining palindrome over ∑={a,b}

Grammar :Production rule (P):

S → aSa rule 1

S → bSb rule 2

S → a/b/ε rule 3

Derive a string "abbabba“

S → aSa

→ abSba from rule 2

→ abbSbba from rule 2

→ abbabba from rule 3 (Required string)

Example 5:

Construct a CFG for the language L = $a^n b^{2n}$ where n>=1,over ∑={a,b}

Grammar :Production rule (P):

S → aSbb rule 1

S → abb rule 2

Derive a string " aabbbb “

S → aSbb from rule 1

→ aabbbb from rule 2 (Required string)

Derivations

- Starting with the start symbol, non-terminals are rewritten using productions until only terminals remain.
- Any terminal sequence that can be generated in this manner is syntactically valid.
- If a terminal sequence can’t be generated using the productions of the grammar it is invalid (has syntax errors).
- The set of strings derivable from the start symbol is the language of the grammar (sometimes denoted L(G)).
- Derivation is a sequence of production rules.
- It is used to get the input string through these production rules.

- During parsing, we need to take the following two decisions.

1. Need to decide the non-terminal which is to be replaced.
2. Need to decide the production rule by which the non-terminal will be replaced.

- Based on the following 2 derivations, We have two options to decide which non-terminal to be placed with production rule .

1. Left most Derivation(LMD)
2. Right most Derivation(RMD)

- To illustrate a derivation, we can draw a derivation tree (also called a parse tree)

Left most Derivation

- 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.
- Leftmost non-terminal is always expanded.

Example:

E = E + E Rule1

E = E - E Rule2

E = a | b Rule3

The leftmost derivation is:

W= a - b + a

E = E + E

E = E - E + E

E = a - E + E

E = a - b + E

E = a - b + a

Rightmost Derivation

- 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 left.
- Rightmost non-terminal is always expanded.

Example:

E = E + E Rule1

E = E - E Rule2

E = a | b Rule3

The rightmost derivation is:

W=a - b + a

E = E - E

E = E - E + E

E = E - E + a

E = E - b + a

E = a - b + a

Parse tree

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parsing, the string is derived using the start symbol.
- The root of the parse tree is that start symbol.
- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

Example:

Grammar G :

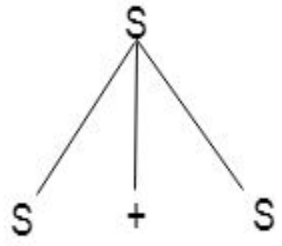
S → S + S | S * S

S → a|b|c

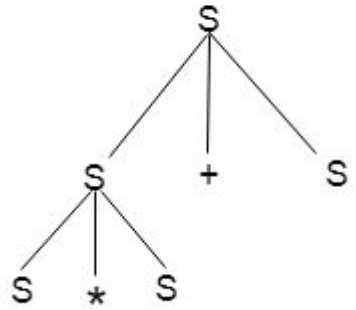
Input String : W=a * b + c

Parse Tree for Left most Derivation

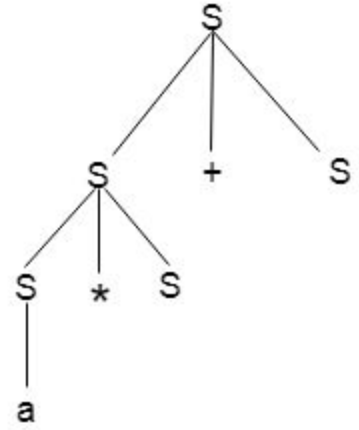
Step 1:



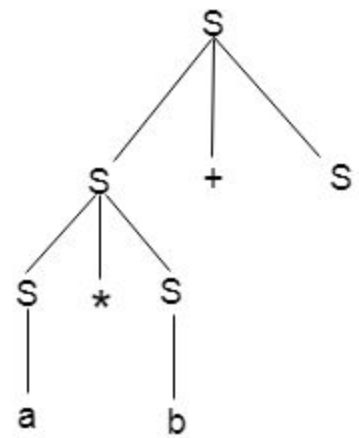
Step 2:



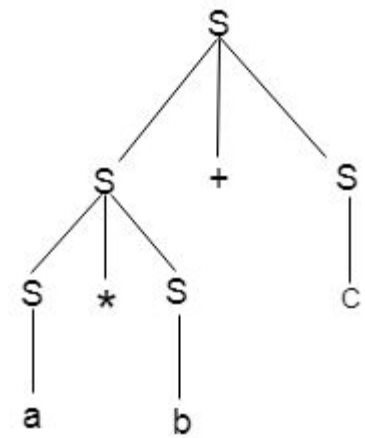
Step 3:



Step 4:



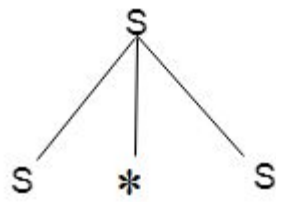
Step 5:



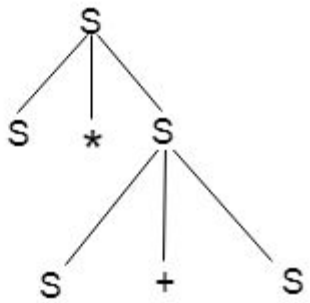
Input String : W=a * b + c

Parse Tree for Right most Derivation

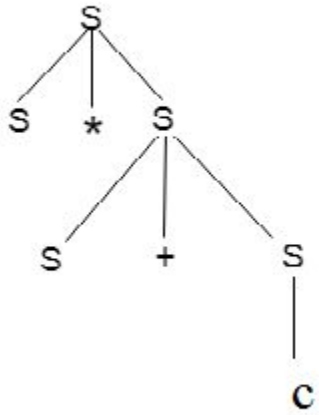
Step 1:



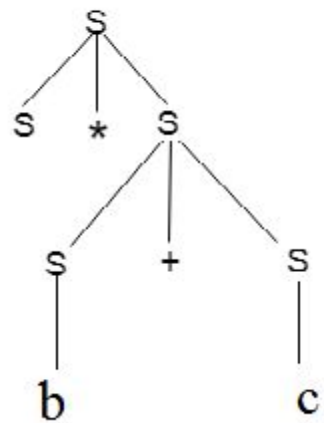
Step 2:



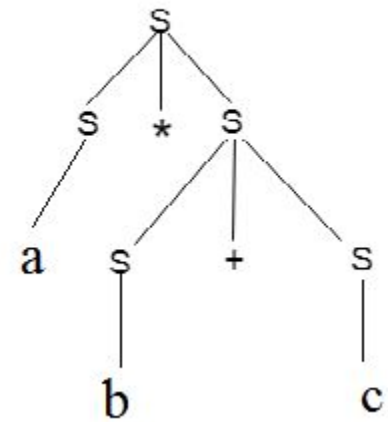
Step 3:



Step 4:



Step 5:



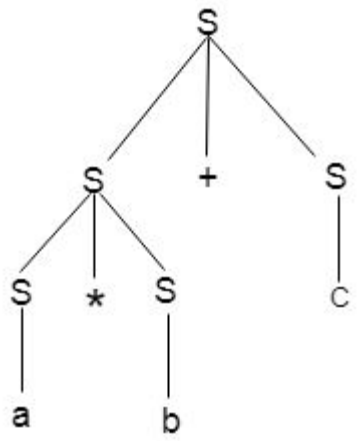
Ambiguous grammar

Ambiguity

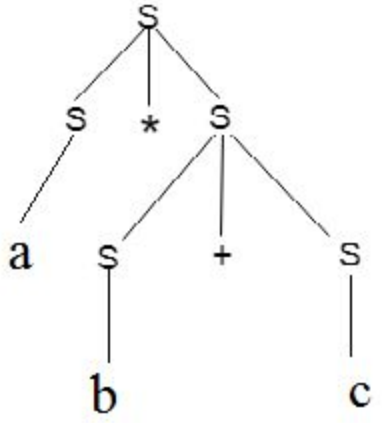
- A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string.

Example1: Input String : W=a * b + c

Parse Tree for Left most Derivation



Parse Tree for Right most Derivation



Contd…

- If the grammar has ambiguity then it is not good for a compiler construction.
- No method can automatically detect and remove the ambiguity but you can remove ambiguity by re-writing the whole grammar without ambiguity.

Example 1: Check whether the given grammar G is ambiguous or not.

E → E + E

E → E - E

E → id

First Leftmost derivation

E → E + E

→ id + E

→ id + E - E

→ id + id - E

→ id + id- id

Second Leftmost derivation

E → E - E

→ E + E - E

→ id + E - E

→ id + id - E

→ id + id - id

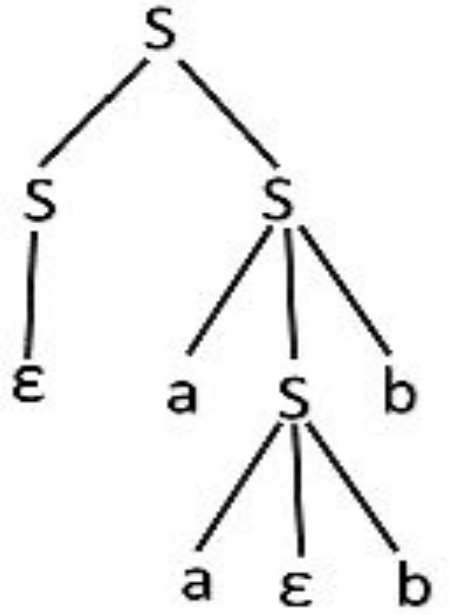
Since there are two leftmost derivation for a single string "id + id - id", the grammar G is ambiguous.

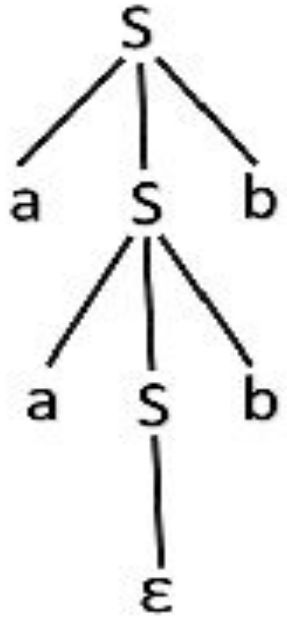
Example 2: Check whether the given grammar G is ambiguous or not.

S → aSb | SS

S → ε

Solution: For the string "aabb" the above grammar can generate two parse trees





Since there are two parse trees for a single string "aabb", the grammar G is ambiguous.

Example 3: Show that the given Expression grammar is ambiguous.

Input Grammar:

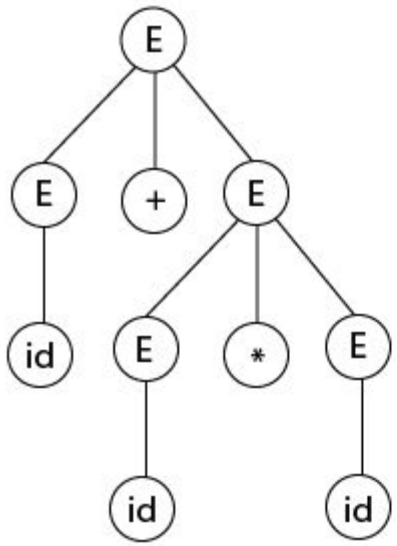
$E \rightarrow E * E$

$E \rightarrow E + E$

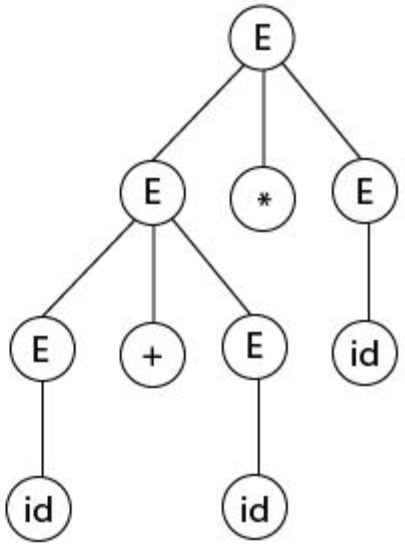
$E \rightarrow id$

Solution:

- Let us derive the string "id + id * id"



Parse tree 1



Parse tree 2

As there are two different parse tree for deriving the same string "id + id * id", the given grammar is ambiguous.

Handling Ambiguity :

1. Precedence –

If different operators are used, we will consider the precedence of the operators. The three important characteristics are :

- The level at which the production is present denotes the priority of the operator used.
- The production at **higher levels** will have **operators with less priority**. In the parse tree, the nodes which are at top levels or close to the root node will contain the lower priority operators.
- The production at **lower levels** will have operators with **higher priority**. In the parse tree, the nodes which are at lower levels or close to the leaf nodes will contain the higher priority operators.

2. Associativity –

If the same precedence operators are in production, then we will have to consider the associativity.

- If the associativity is left to right, then we have to prompt a left recursion in the production. The parse tree will also be left recursive and grow on the left side.
  +, -, *, / are left associative operators.
- If the associativity is right to left, then we have to prompt the right recursion in the productions. The parse tree will also be right recursive and grow on the right side.
  ^ is a right associative operator.

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

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

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

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

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

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

$$S \rightarrow Aa \mid b$$

$$\Lambda \; \rightarrow bd\Lambda' \mid \Lambda'$$

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

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:

  $S \rightarrow iEtSS' \mid a$

  $S' \rightarrow eS \mid \in$

  $E \rightarrow b$

Ex 2:

S -> bSSaaS | bSSasb | bSb | a

- **Bs** is the common prefix here so converting the above grammar to Left Factored Grammar-

  S -> BSS’ | a

  S’ -> SaaS | SaSb | B

- Sa is the common prefix here so converting the above grammar to Left Factored Grammar-

  S’ -> SaS’’ | B

  S’’ -> aS | Sb

Final Grammar: S -> BSS’ | a

S’ -> SaS’’ | B

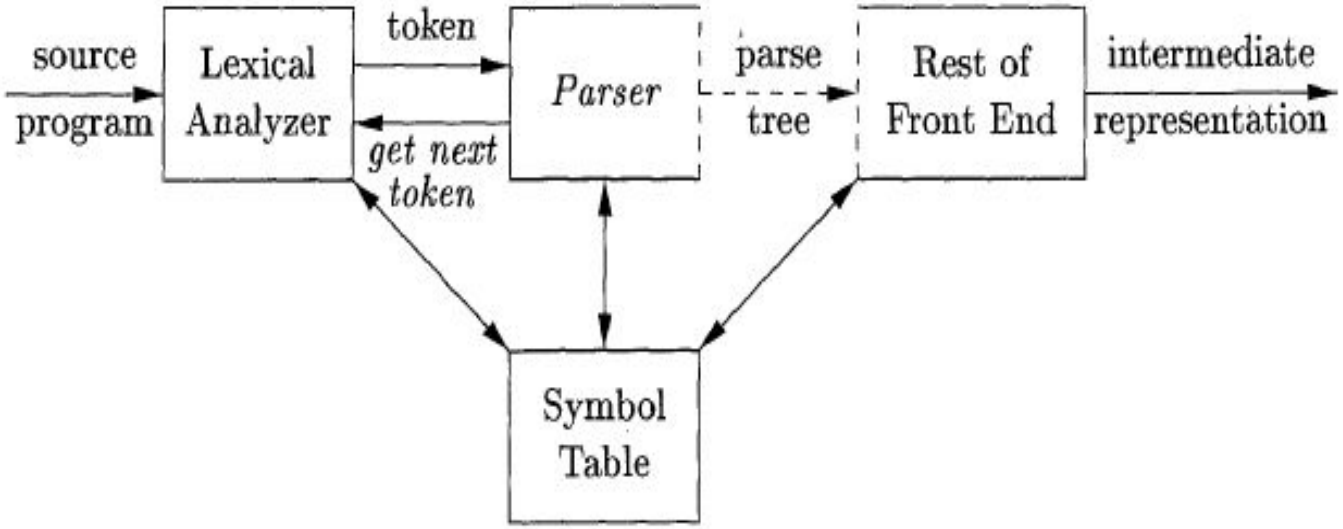
S’’ -> aS | Sb

Syntax Analysis

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

The role of the parser

- The parser obtains a string of tokens from the lexical analyzer, and verifies that the string of token names can be generated by the grammar for the source language.
- The parser reports any syntax errors.
- The parser constructs a parse tree and passes it to the rest of the compiler for further processing.



Syntax 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

1. **Panic- Mode Recovery: On discovering an error, the parser discards input symbols one at a time** until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as ; or }, whose role in the source program is clear and unambiguous.
2. **Phase- Level Recovery:** On discovering an error, a parser may perform local correction on the remaining input; that is, **it may replace a prefix of the remaining input** by some string that allows the parser to continue. A typical local correction is to **replace a comma by a semicolon**, **delete an extraneous semicolon**, or **insert a missing semicolon**.
3. **Error Productions:** By **anticipating common errors** that might be encountered, we can **augment the grammar** for the language at hand with productions that generate the erroneous constructs. A parser constructed from a grammar augmented by these error when an error production is used during parsing.

4. Global Correction: There are algorithms for choosing a minimal sequence of changes to obtain a globally least- cost correction. Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y, such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible.

Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.

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, $Y_1 \ldots Y_{i-1} \Rightarrow^* \in$. 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

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

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

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

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. $\beta \Rightarrow^* \in$), then everything in FOLLOW(A) is in FOLLOW(B).

Example

Consider the grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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

After applying FIRST rules over the grammar

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

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

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

Another Example of FIRST and FOLLOW

- Grammar G

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

  $A \rightarrow da \mid BC$

  $B \rightarrow g \mid \epsilon$

  $C \rightarrow h \mid \epsilon$

FIRST(S) will be calculated from 3 productions:

1. S -> ACB 2. S -> CbA 3. S -> Ba

1. **S -> ACB**

As FIRST(S) is **A**, place both FIRST values of A here-

a. S -> dCB, from here **FIRST(S) = {d}**

b. S -> εCB => **S -> CB**, from here FIRST (S) is **C** so place both FIRST values of C in this new production

(i). S -> hB, so from here **FIRST(S) = {h}**

(ii) S -> εB => **S -> B**, from here FIRST(S) is B so place both FIRST values of B in this new production

- S -> g, so from here **FIRST(S) = {g}**
- S -> ε, so from here **FIRST(S) = {ε}**

Combining all values of **FIRST(S) = {d, h, g, ε}** **—(x)**

2. S -> CbA

As FIRST(S) is **C**, place both FIRST values of C here-

a. S -> hbA, so from here **FIRST(S) = {h}**

b. S -> εbA => **S -> bA**, so from here **FIRST(S) = {b}**

FIRST(S) = {h, b} — (y)

3. S -> Ba

As FIRST(S) is **B**, place both FIRST values of B here-

a. S -> ga, so from here **FIRST(S) = {g}**

b. S -> εa => **S -> a**, so from here **FIRST(S) = {a}**

FIRST(S) = {g, a} — (z)

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

From equation (x), (y) and (z)-

FIRST (S) = {d, h, g, ε, b, a}

Production Rule	FIRST	FOLLOW
S -> ABCD	{b}	{$}
A -> b	{b}	{c}
B -> c	{c}	{d}
C -> d	{d}	{e}
D -> e	{e}	{$}

Production Rule	FIRST	FOLLOW
S -> ABCDE	{a, b, c}	{$}
A -> a | ε	{a, ε}	{b, c}
B -> b | ε	{b, ε}	{c}
C -> c	{c}	{d, e, $}
D -> d | ε	{d, ε}	{e, $}
E -> e | ε	{e, ε}	{$}

Production Rule	FIRST	FOLLOW
S -> Bb | Cd	{a, b, c, d}	{$}
B -> aB | ε	{a, ε}	{b}
C -> cC | ε	{c, ε}	{d}

Production Rule	FIRST	FOLLOW
E -> TE’	{id, (}	{$,)}
E’ -> +TE’ | ε	{+, ε}	{$,)}
T -> FT’	{id, (}	{+, $,)}
T’ -> *FT’ | ε	{*, ε}	{+, $,)}
F -> (E) | id	{id, (}	{*, +, $,)}

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.

Parsing

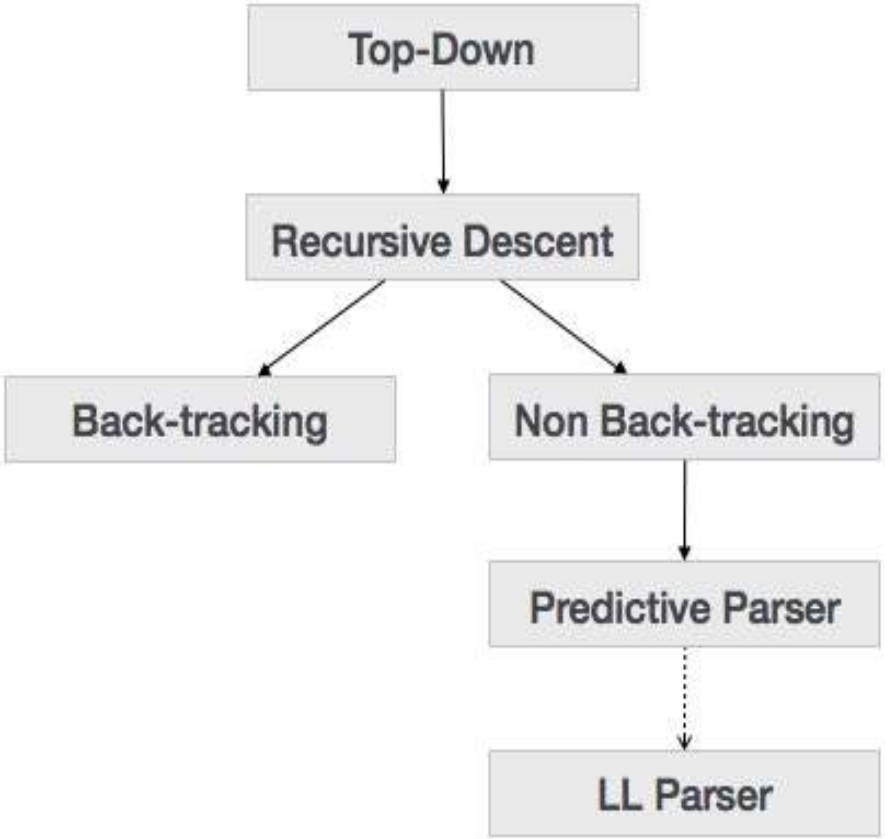
- *Universal* (any C-F grammar)
  - Cocke-Younger-Kasimi
  - Earley
- *Top-down* (C-F grammar with restrictions)
  - Recursive descent (predictive parsing)
  - LL (Left-to-right, Leftmost derivation) methods
- *Bottom-up* (C-F grammar with restrictions)
  - Operator precedence parsing
  - LR (Left-to-right,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

Top Down Parser

Top-Down Parser

Recursive Descent Parsing

- Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.
- This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.



Example: Recursive Descent Parsing

E → i E‘

E' → + i E' | ε

- Main()
  {
  E(); // Start parsing from E
  if (input == '$') // If we reach end of input
  Parsing Successful;
  }

- **Function E()**

```
E()
  {
  if (input == 'i')       // If the input is 'i' (identifier)
 {
  input++;                // Consume 'i'
  }
  E'();                   // Call E' to check for further expressions
  }
```

- **Function E'()**

```
void E`() {
  if (input == '+') {
  input++;          // Consume the '+'

  if (input == 'i') {
  input++;           // Consume the 'i'
  }

  E`();              // Recursively process more additions
  } else {
  return;           // If no '+', return (ε production)
  }
  }
```

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.

Back-tracking

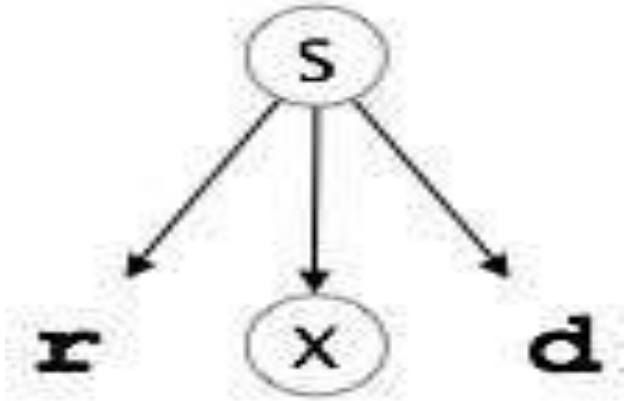
- Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

  S → rXd | rZd

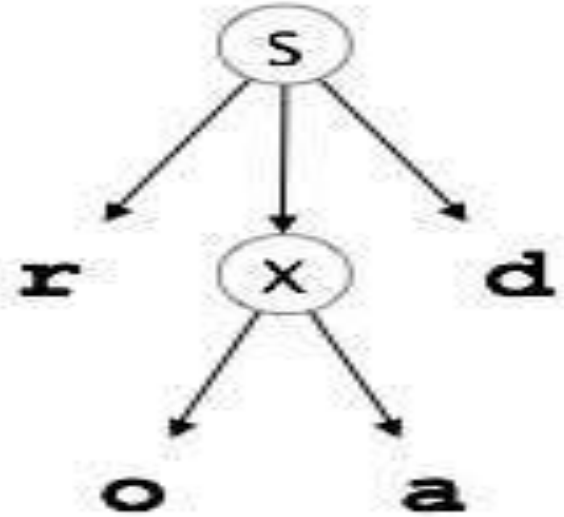
  X → oa | ea

  Z → ai

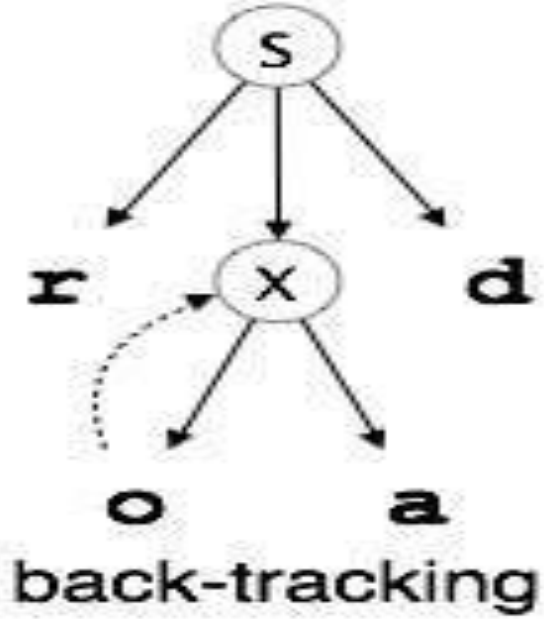
- It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. ‘e’). The parser tries to expand non-terminal ‘X’ and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).
- Now the parser matches all the input letters in an ordered manner. The string is accepted.



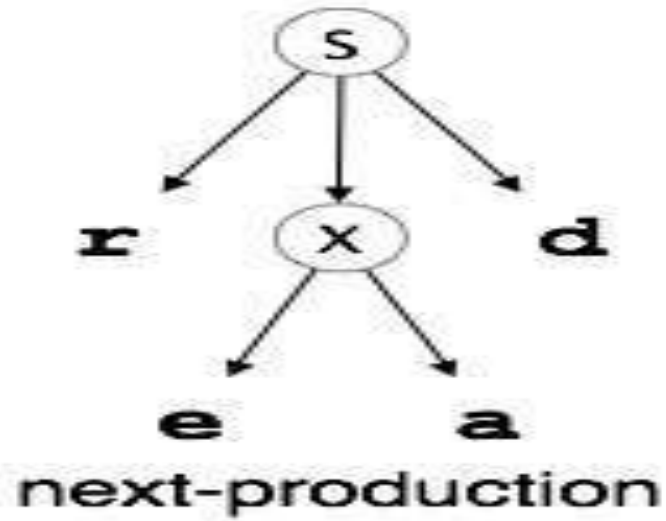
a



b



c

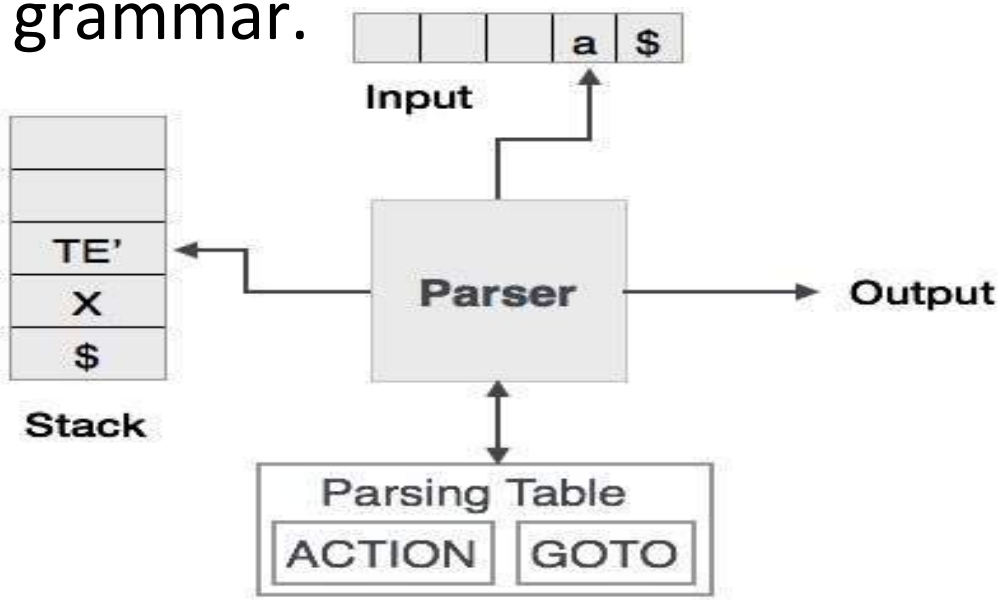


d

Non Back tracking:

Predictive Parser

- Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.
- To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



Transition Diagrams for Predictive Parsers

Consider the grammar:

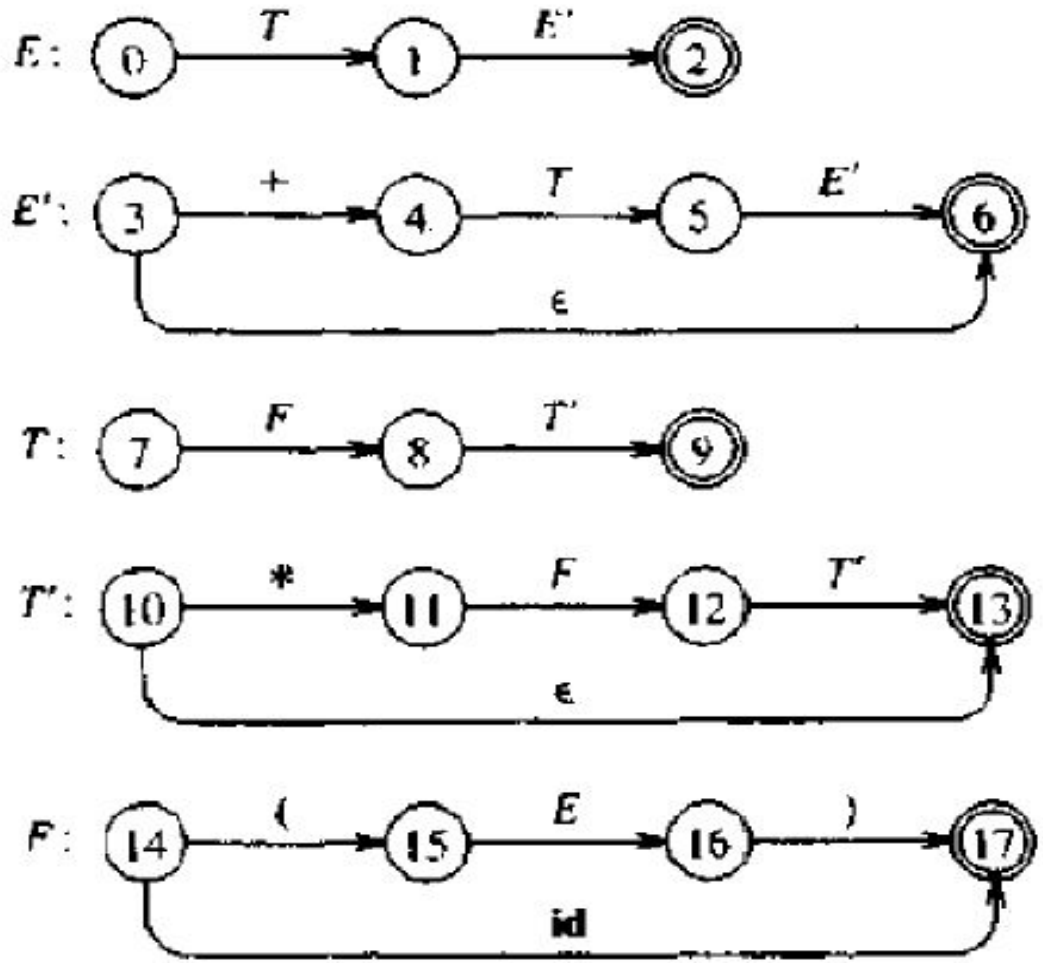
$E \rightarrow TE'$

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

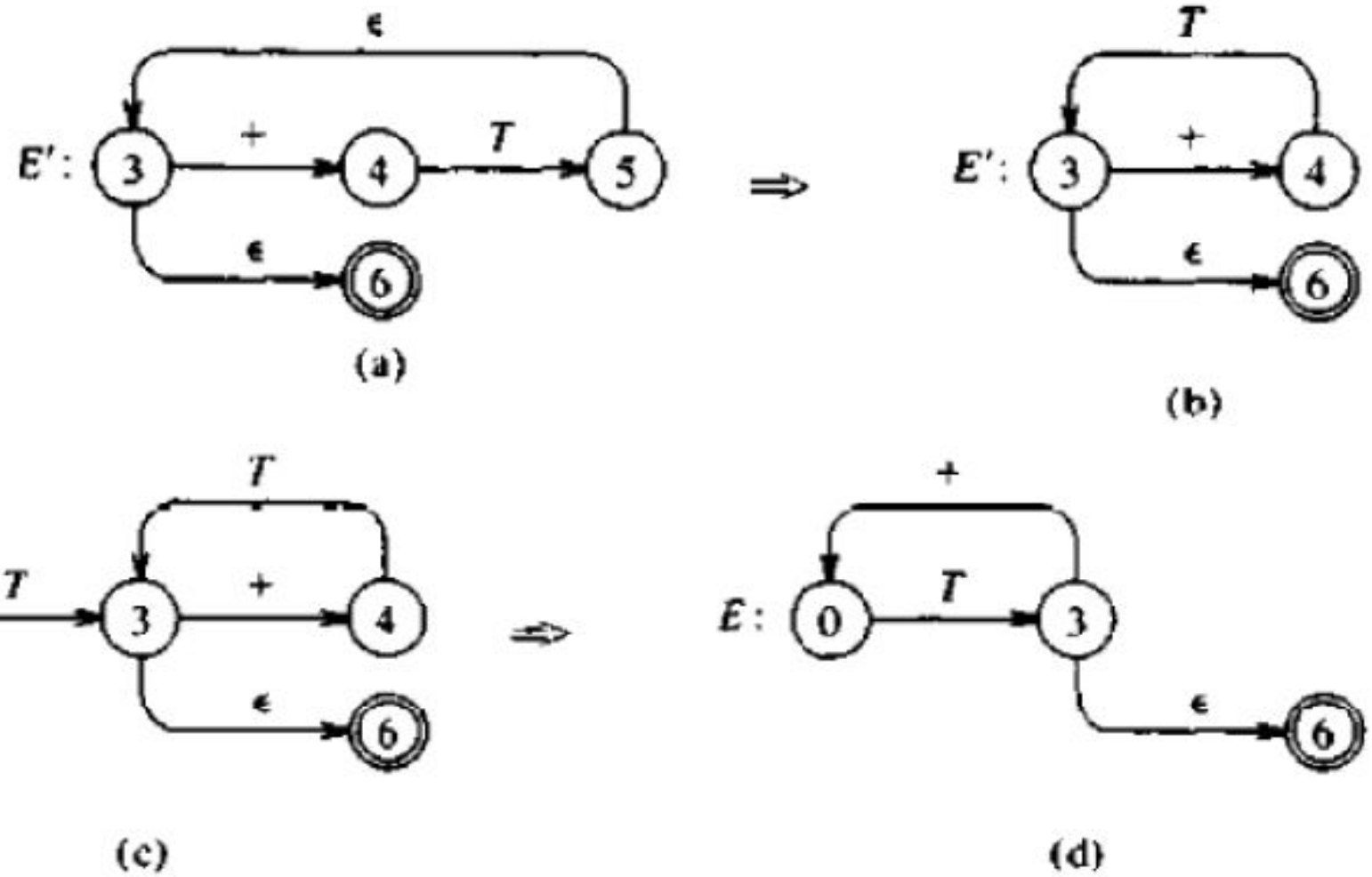
$T \rightarrow FT'$

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

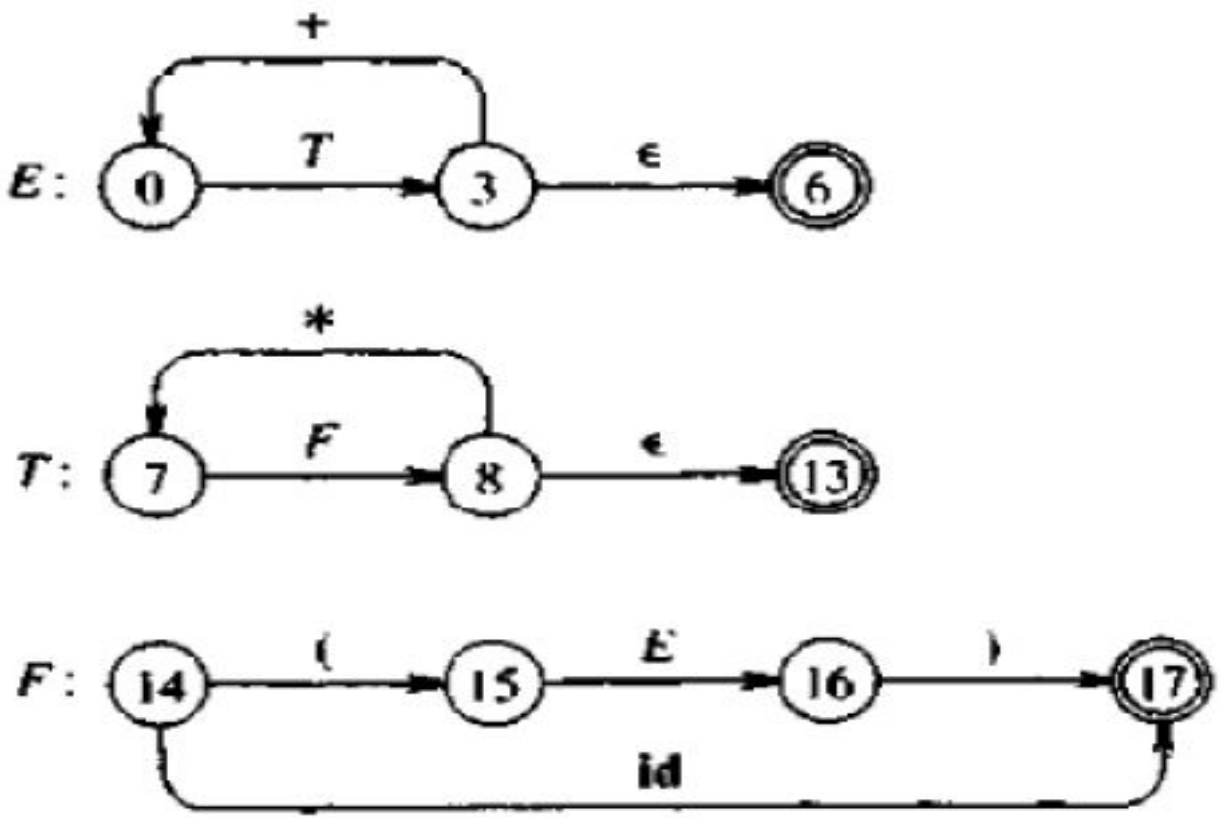
$F \rightarrow (E) \mid \mathbf{id}$



Transition Diagrams for Predictive Parsers

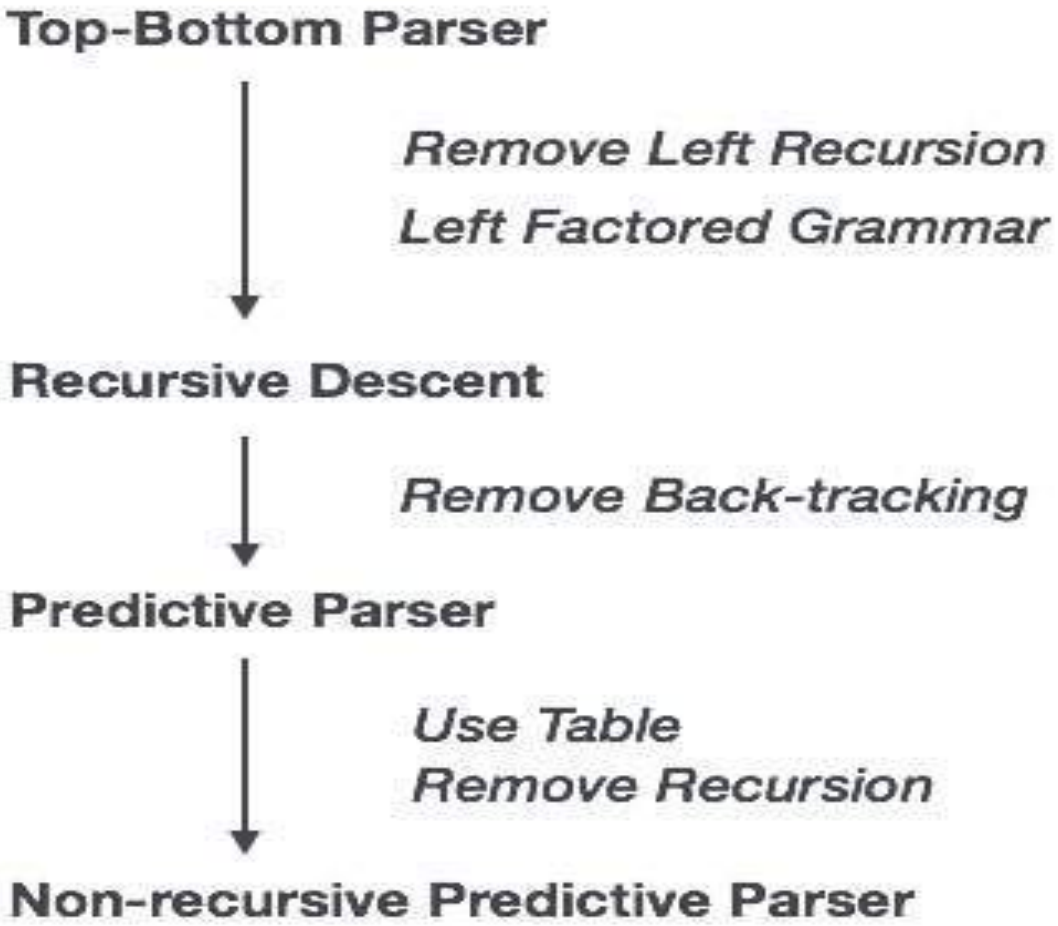


Transition Diagrams for Predictive Parsers



Simplified transition diagrams for arithmetic expressions.

- Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol **$** to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.



- In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

LL Parser

- An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.
- LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).

LL(1) Parser (Top Down Parser)

1. L – Scan input from left to right
2. L – Leftmost derivation
3. 1 – No of lookaheads (how many symbols to see to make a decision)

LL1 Parsing

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

LL Parsing Algorithm

- We may stick to deterministic LL(1) for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not LL(1), then usually, it is not LL(k), for any given k.

Input:

string ω

parsing table M for grammar G

Output:

If ω is in L(G) then left-most derivation of ω,

error otherwise.

Initial State : $S on stack (with S being start symbol)

ω$ in the input buffer

Algorithm:

SET ip to point the first symbol of ω$.
repeat
  let X be the top stack symbol and a the symbol pointed by ip.

  if X∈ Vt or $
    if X = a
      POP X and advance ip.
    else
      error()
    endif

```
 else   /* X is non-terminal */
      if M[X,a] = X → Y1, Y2,... Yk
         POP X
         PUSH Yk, Yk-1,... Y1 /* Y1 on top */
         Output the production X → Y1, Y2,... Yk


else
         error()
      endif
   endif
until X = $     /* empty stack */
```

LL(1) Parsing Table

Example:
E -> E + T | T
T -> T * F | F
F -> (E) | id

Step 1: Remove Left Recursion
E -> TE’
E’ -> +TE’ | ε
T -> FT’
T’ -> *FT’ | ε
F -> (E) | id

Step 2: Already left factored grammar

Step 3:

Production Rule	FIRST	FOLLOW
E -> TE’	{id, (}	{$,)}
E’ -> +TE’ | ε	{+, ε}	{$,)}
T -> FT’	{id, (}	{+, $,)}
T’ -> *FT’ | ε	{*, ε}	{+, $,)}
F -> (E) | id	{id, (}	{*, +, $,)}

Step 4: Parsing Table

	id	**+**	*****	**(**	**)**	**$**
E	E -> TE’			E -> TE’		
E’		E’ -> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’			T -> FT’		
T’		T’ -> ε	T’ -> *FT’		T’ -> ε	T’ -> ε
F	F -> id			F -> (e)		

Parse the string: id + id * id

Stack	**Input String**	**Actions**
$ E	id + id * id $	E -> TE’
$ E’ T	id + id * id $	T -> FT’
$ E’ T’ F	id + id * id $	F -> id
$ E’ T’ id	id + id * id $	Pop id
$ E’ T’	+ id * id $	T’ -> ε
$ E’	+ id * id $	E’ -> + TE’
$ E’ T +	+ id * id $	Pop +
$ E’ T	id * id $	T -> F T’
$ E’ T’ F	id * id $	F -> id

Stack	Input String	Actions
$ E’ T’ F	id * id $	F -> id
$ E’ T’ id	id * id $	Pop id
$ E’ T’	* id $	T’ -> *F T’
$ E’ T’ F *	* id $	Pop *
$ E’ T’ F	id $	F -> id
$ E’ T’ id	id $	Pop id
$ E’ T’	$	T’ -> ε
$ E’	$	E’ -> ε
$	$	Accept

Ques: Draw LL(1) parsing table for the following grammar:
S -> aABb
A -> c | ε
B -> d | ε
Also check validity of the string: acdb

Step 1: No Left recursion
Step 2: Already left factored grammar
Step 3: Compute First and Follow
First (S) = {a}
First (A) = { c, ε}
First (B) = { d, ε}

Follow(S) = {$}
Follow (A) = { d, b}
Follow(B) = {b}

Step 4: LL(1) Parsing Table:

	a	b	c	d	$
S	S -> aABb				
A		A -> ε	A -> c	A -> ε	
B		B -> ε		B -> d	

String Validation:

Stack	Input String	Action
$ S	acdb$	S -> aABb
$ bBAa	acdb$	Pop a
$ bBA	cdb$	A -> c
$ bBc	cdb$	Pop c
$ bB	db$	B -> d
$ bd	db$	Pop d
$ b	b$	Pop b
$	$	Accept

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

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

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

LL(1) Grammars are ambiguous

- Consider the Grammar

  $S \rightarrow iEtSS' \mid a$

  $S' \rightarrow eS \mid \in$

  $E \rightarrow b$

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).**

Non-LL(1) Examples

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

Error Recovery in Predictive Parsing

- Predictive parser performs Left most derivative while parsing the given sentence. Now the given sentence may be a valid sentence or an invalid sentence with respect to the specified grammar. An error is detected during the predictive parsing when the terminal on top of the stack does not match the next input symbol, or when nonterminal X on top of the stack, then the present input symbol is a, and the parsing table entry M [X, a] is considered empty.
- **It is also possible that An error may occur while performing predictive parsing (LL(1) parsing)**

1. When the terminal symbol is at top of the stack and it does not match the current input symbol.

2. In LL(1) parsing If the top of the stack is a non-terminal A, then the present input symbol is **a**, and the parsing table entry M [A, a] is considered empty.

- **Error Recovery Techniques:**
- **Panic-Mode Error Recovery:** In Panic-Mode Error Recovery the technique is skipping the input symbols until a synchronizing token is found.
- **Phrase-Level Error Recovery**: Each empty entry in the parsing table is filled with a pointer to a specific error routing take care of that error case**.**

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***.

- 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.

- 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	$F \to$ **id**	synch	synch	$F \to (E)$	synch	synch

Synchronizing tokens added to parsing table.

- 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.

Solution 1:

- 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

Solution 2:

$E \rightarrow T\,E'$

$E' \rightarrow +\,T\,E' \mid \in$

$T \rightarrow F\,T'$

$T' \rightarrow *\,F\,T' \mid \in$

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

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***

- 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'	$	$T' \rightarrow \in$
$	$	$E' \rightarrow \in$