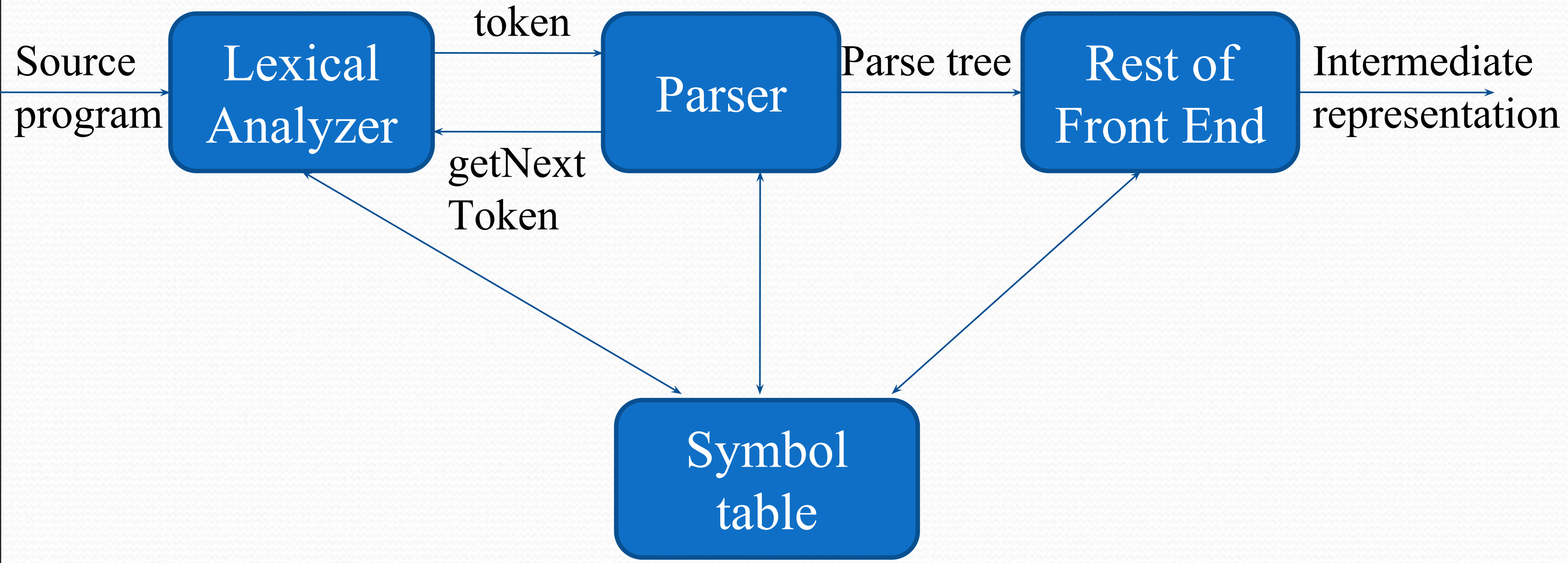
Syntax Analysis

Outline

- Introduction
- Context free grammars
- Writing a grammar
- Top down parsing
- Bottom up parsing

The role of parser



The role of parser

- Parser obtains a string of tokens from the analyser and verifies that the string of token names can be generated by the grammar for the source language.
- The parser constructs a parse tree and passes it to the rest of the compiler for further processing.
- There are three general types of parsers:
  - Universal:- Cocke-Younger-Kasami & Earley’s algorithm
  - top-down :- build parse tees from the root to the leaves. (LL)
  - bottom-up:- start from the leaves and work their way up to the root. (LR)
- In either case the input to the parser is scanned from left to right, one symbol at a time.

Uses of grammars

E-> expressions, T-> terms, F-> factors

Left-recursive grammar- suitable for bottom-up parsing

E -> E + T | T

T -> T * F | F

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

Non-left-recursive grammar- suitable for top-down parsing

E -> TE'

E' -> +TE' | Ɛ

T -> FT'

T' -> *FT' | Ɛ

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

Error handling

- Common programming errors
  - Lexical errors
  - Syntactic errors
  - Semantic errors
  - Logical 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-recover strategies

- **Panic mode recovery**
  - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- **Phrase level recovery**
  - Replacing a prefix of remaining input by some string that allows the parser to continue
- **Error productions**
  - Augment the grammar with productions that generate the erroneous constructs
- **Global correction**
  - Choosing minimal sequence of changes to obtain a globally least-cost correction

Context-Free grammars

- Grammars are used to describe the syntax of programming language constructs like expressions and statements.
- A syntactic variable *stmt*-> statements and variable *expr*-> expressions, the production

  stmt -> **if** (*expr*) *stmt* **else** *stmt*

- A context-free grammar consists of terminals, nonterminals, a start symbol and productions.

Context-Free grammars

- Terminals:- basic symbols from which strings are formed. It is a synonym for token name. Ex: if, else, (,)
- Nonterminals:- syntactic variables that denote a set of strings. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation. Ex: *stmt, expr*
- Start symbol:- a nonterminal, start symbol and set of strings it denotes is the language generated by the grammar. The productions for the start symbol are listed first. Ex: LHS stmt symbol

Context-Free grammars

- Productions:- the productions of a grammar specify the manner in which the terminals and the nonterminals can be combined to form strings. Each production consists of;
  - A nonterminal called head or left side of the production; this production defines some of the strings denoted by the head.
  - The symbol -> , sometimes : : = has been used in the place of the arrow
  - A body or right side consisting of zero or more terminals and nonterminals.

Context-Free grammars

Grammar for simple arithmetic expressions

expression -> expression + term

expression -> expression – term

expression -> term

term -> term * factor

term -> term / factor

term -> factor

factor -> (expression)

factor -> **id**

id + - * / ( ) -> terminals

expression, term, factor -> nonterminals

expression -> start symbol

Derivations

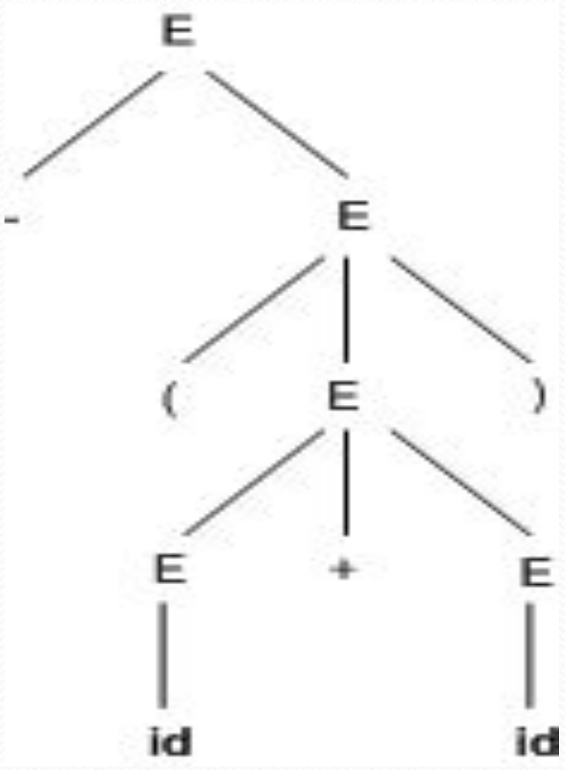
- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
  - E -> E + E | E * E | -E | (E) | **id**
  - Derivations for **–(id+id)**
    - E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**) -Left
    - E=> - E => -(E) =>-(E+E) => -(E+id) =>-(id+id) –Right

Parse trees

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

Parse trees and Derivations

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



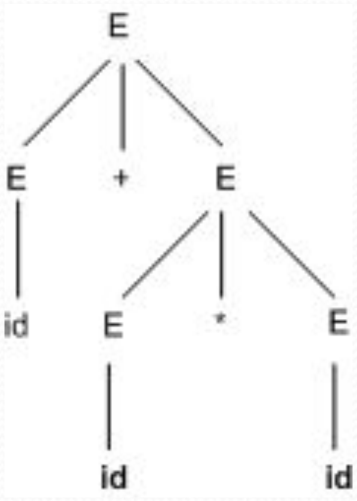
Ambiguity

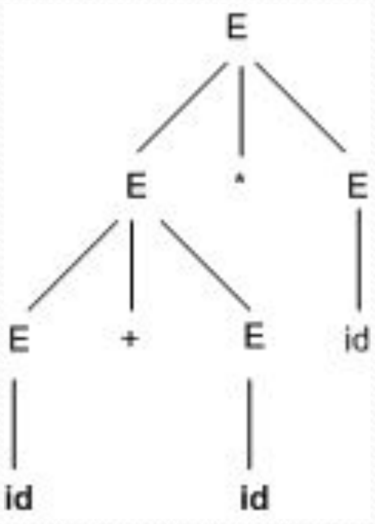
- For some strings there exist more than one parse tree
- Or more than one leftmost derivation
- Or more than one rightmost derivation
- Example: id+id*id   E -> E + E | E * E | -E | (E) | **id**

```
E => E + E,                 E => E * E
  => id + E               => E + E * E
  => id + E * E             => id +  E * E
  => id + id * E            => id + id * E
  => id + id * id             => id + id * id
```

Ambiguity

- Example: id+id*id
- Expression like a + b * c as a + (b * c) rather than (a + b) *c





Verifying the Language

- A grammar G generates a language L has two parts:
  - Every string generated by G in L
  - Conversely that every string in L and indeed be generated by G.

Example: Grammar - S => (S) S | Ɛ

Basis : The basis is n=1. The only string of terminals derived from S in one step is the Ɛ –empty string, which is balanced.

Induction: s => (S)S =>* (x)S =>* (x)y.

CFG versus Regular Expressions

The regular expression (a|b)*abb and the grammar describes the same language , the set of strings of a’s and b’s ending in abb.

A_0 -> aA_0 | bA_0 | aA_1
A_1 -> bA_2
A_2 -> bA_3
A_3 -> ε

Example

Consider the context-free grammar:

S -> S S + | S S * | a and the string aa+a*.

a)Give a leftmost derivation of the string

b)Give a rightmost derivation of the string

c)Give a parse tree for the string

d)Is grammar ambigous or unambiguos? Justify the answer.

e)Describe the language generated by this grammar.

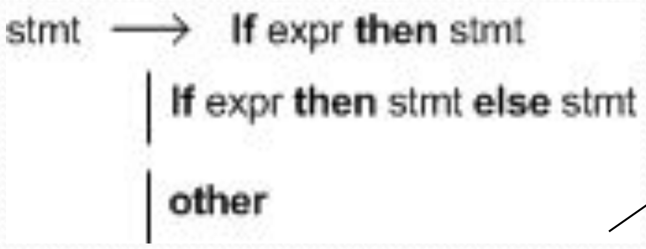
Leftmost derivation: s=>ss*=>ss+s*=>as+s*=>aa+s*=>aa+a*

Rightmost derivation: s=>ss*=>sa*=>ss+a*=>sa+a*=>aa+a*

Lexical Versus Syntactic Analysis

- Separate the syntactic structure of a language into lexical and non lexical parts provides a convenient of modularizing the front end of a compiler into two manageable- sized components.
- The lexcial rules are simpler, does not require notations as grammars.
- Regular expressions are easier to understand than grammars.
- More efficient lexical analyzers can be consrtucted automatically from regular expressions than from grammars.

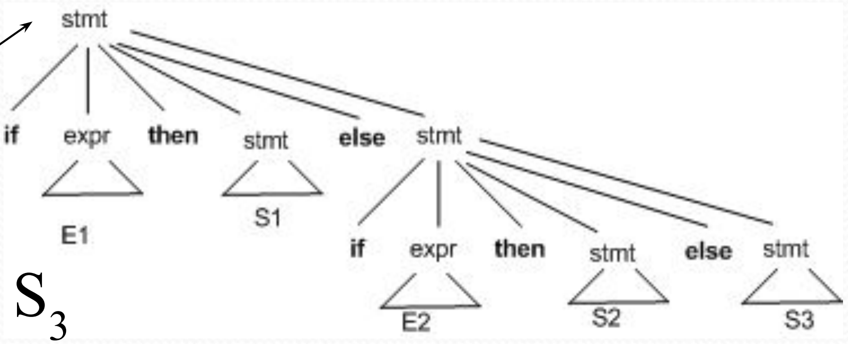
Elimination of ambiguity



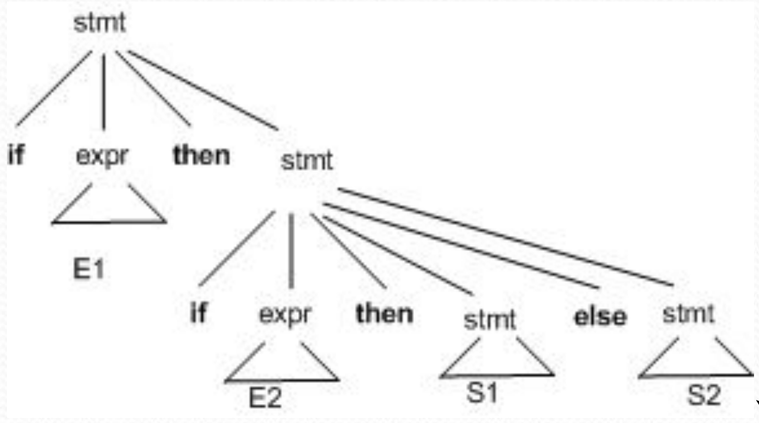
stmt → **If** expr **then** stmt

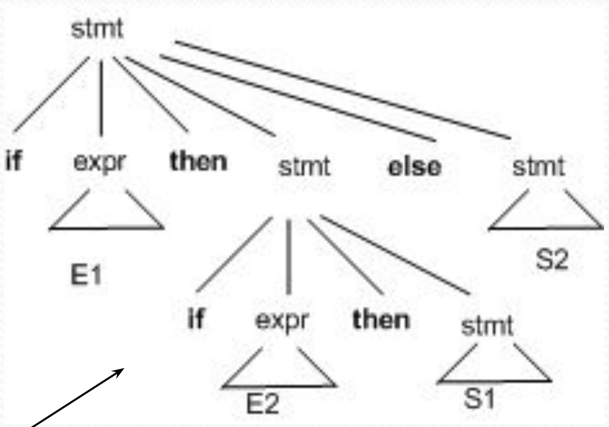
| **If** expr **then** stmt **else** stmt

| **other**



if E_1 then S_1 else if E_2 then S_2 else S_3

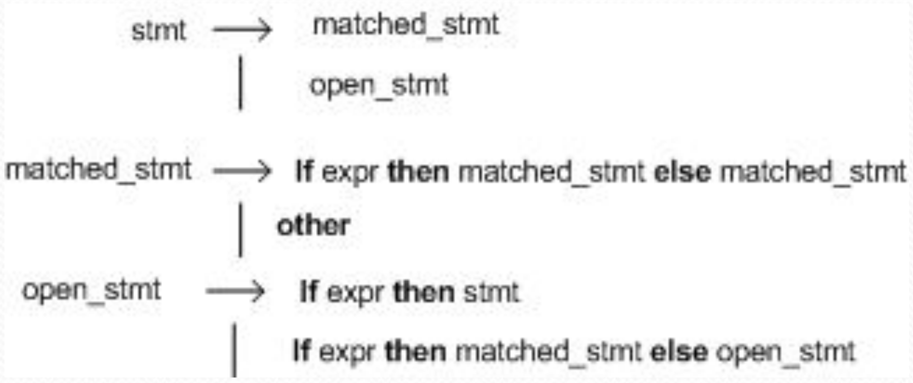




if E_1 then if E_2 then S_1 else S_2

Elimination of ambiguity (cont.)

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



```
stmt          →  matched_stmt
              |  open_stmt
matched_stmt  →  If expr then matched_stmt else matched_stmt
              |  other
open_stmt     →  If expr then stmt
              |  If expr then matched_stmt else open_stmt
```

Elimination of left recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.
- Top down parsing methods cant handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:
    - $A \to A\,\alpha | \beta$
  - We may replace it with
    - $A \to \beta\, A'$
    - $A' \to \alpha\, A' \mid \varepsilon$

Left recursion elimination (cont.)

- There are cases like following
  - S -> Aa | b
  - A -> Ac | Sd | ε
- Left recursion elimination algorithm:
  - Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
  - for (each i from 1 to n) {
    - for (each j from 1 to i-1) {
      - Replace each production of the form $A_i \rightarrow A_j\,\gamma$ by the production $A_i \rightarrow \delta 1\,\gamma \mid \delta 2\,\gamma \mid \ldots \mid \delta_k\,\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all current A_j productions
      - }
      - Eliminate left recursion among the Ai-productions
    - }

Left recursion elimination (cont.)

- There are cases like following
  - S -> Aa | b
  - A -> Ac | Sd | ε

  Applying the algorithm,
  - There is no change in first production, as it is not left recursive
  - Second production , A-> S d should be replaced as
  -    A -> Ac | A a d | b d | ε

  After removing left recursion
  - S -> A a | b
  - A’ -> b d A’ | A’
  - A’ -> c A’ | a d A’ | ε

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Consider following grammar:
  - Stmt -> **if** expr **then** stmt **else** stmt

    | **if** expr **then** stmt
- On seeing input **if** it is not clear for the parser which production to use
- We can easily perform left factoring:
  - If we have A->αβ1 | αβ2 then we replace it with
    - A -> αA'
    - A' -> β1 | β2

Left factoring (cont.)

- Algorithm
  - For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If α <> ε, then replace all of A-productions A-> $\alpha \beta 1 \mid \alpha \beta 2 \mid \ldots \mid \alpha \beta n \mid \gamma$ by
    - A -> α A' | γ
    - A' -> $\beta 1 \mid \beta 2 \mid \ldots \mid \beta n$
- Example:
  - S -> i E t S | i E t S e S | a
  - E -> b

Left factoring (cont.)

- Example:
  - S -> i E t S | i E t S e S | a
  - E -> b

  After left factoring, the grammar becomes:

  S -> i E t S S'| a

  S -> e S | ε

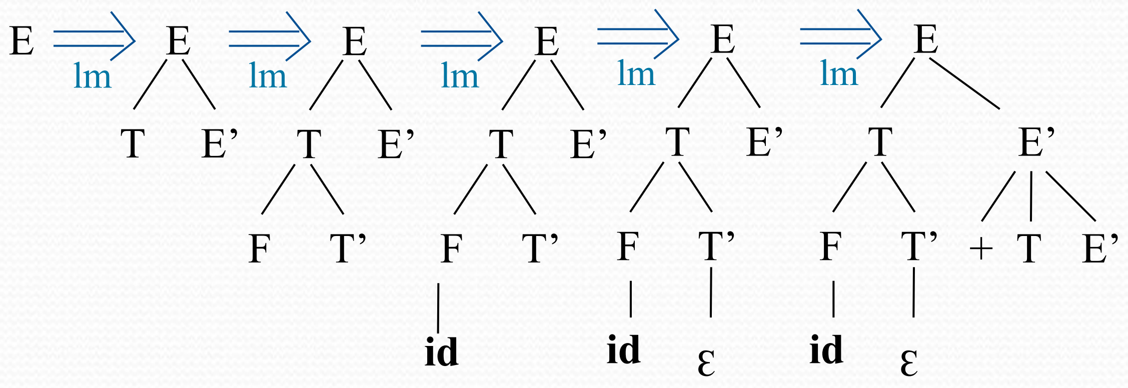
  E -> b

Top Down Parsing

Introduction

- A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right
- It can be also viewed as finding a leftmost derivation for an input string
- Example: id+id*id

E -> TE'
E' -> +TE' | Ɛ
T -> FT'
T' -> *FT' | Ɛ
F -> (E) | **id**



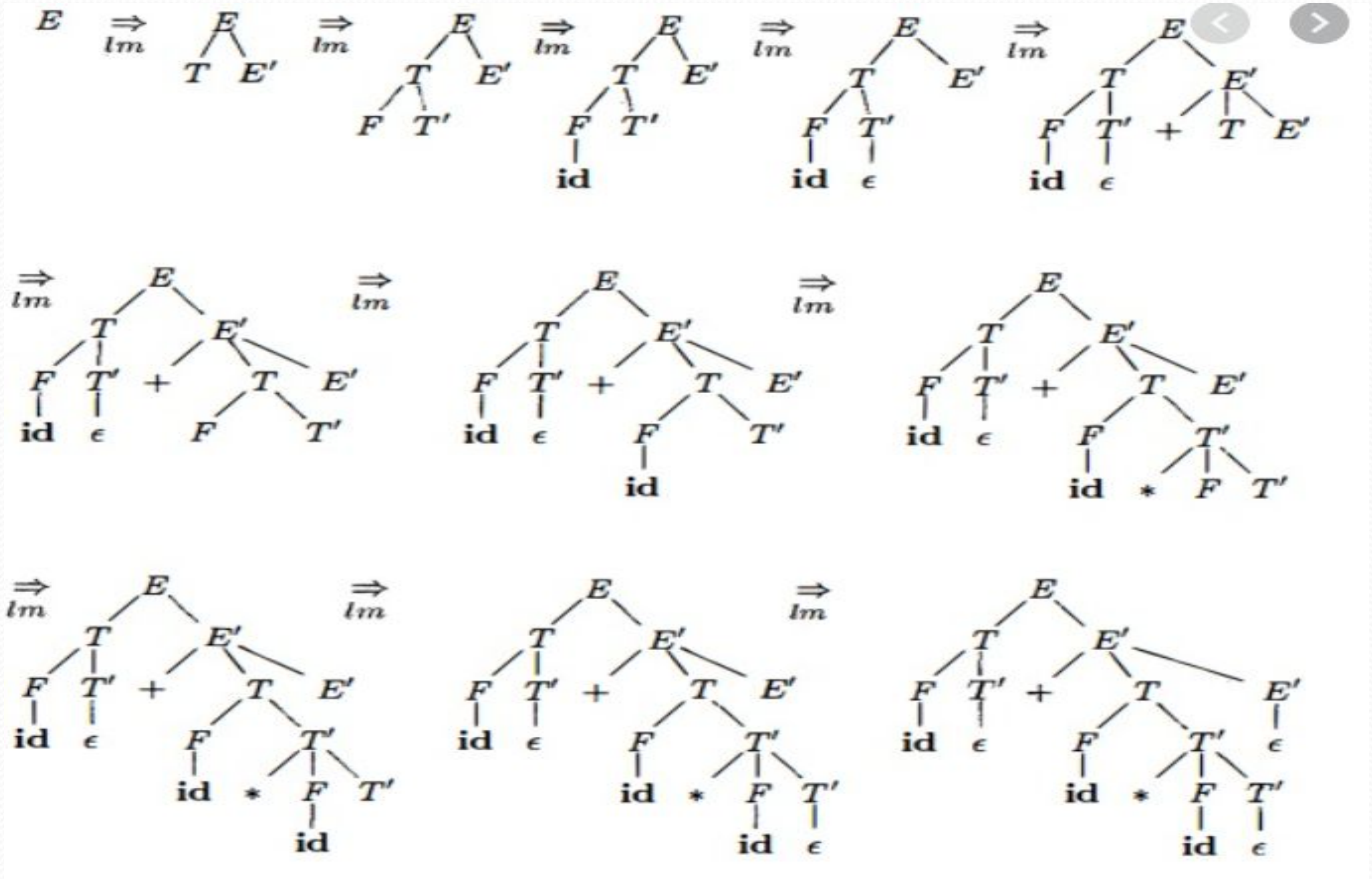


Figure 4.12: Top-down parse for **id** + **id** ∗ **id**

Type of Top-down parser

- Recursive-Descent Parsing
  - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
  - It is a general parsing technique, but not widely used.
  - Not efficient
- Predictive Parsing
  - no backtracking
  - efficient
  - needs a special form of grammars (LL(1) grammars).
  - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
  - Non-Recursive Predictive Parser is also known as LL(1) parser.

Recursive descent parsing

- Consists of a set of procedures, one for each nonterminal.
- Execution begins with the procedure for start symbol
- A typical procedure for a non-terminal

```
void A() {
    choose an A-production, A->$X_1X_2..X_k$
    for (i=1 to k) {
        if ($X_i$ is a nonterminal
            call procedure $X_i$();
        else if ($X_i$ equals the current input symbol a)
            advance the input to the next symbol;
        else /* an error has occurred */
    }
}
```

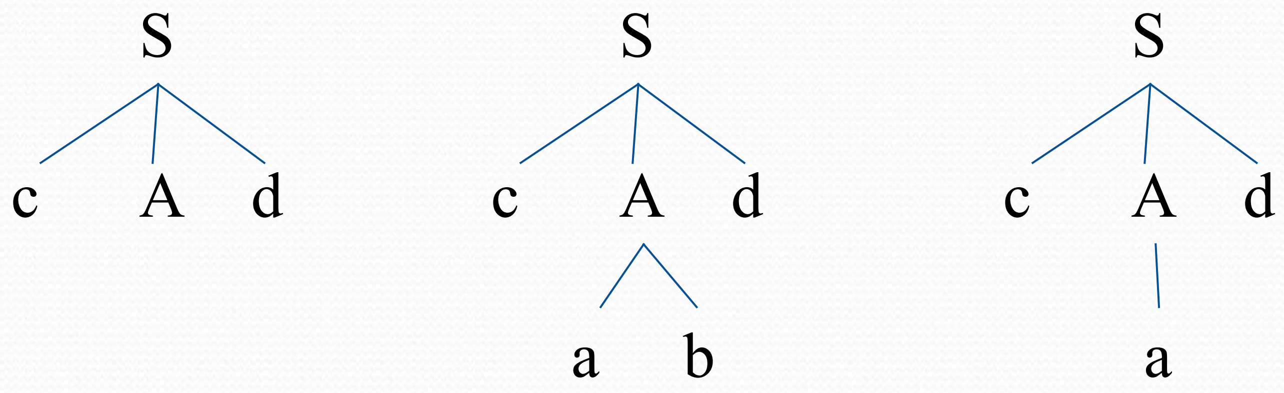
Recursive descent parsing (cont)

- General recursive descent may require backtracking
- The previous code needs to be modified to allow backtracking
- In general form, it can’t choose an A-production easily, so we need to try all alternatives
- If one failed the input pointer needs to be reset and another alternative should be tried
- Recursive descent parsers cant be used for left-recursive grammars

Example

S->cAd
A->ab | a

Input: cad



First and Follow

- Two functions are used in the construction of LL(1) parsing tables:
  - FIRST   FOLLOW
- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.
- if α derives to ε, then ε is also in FIRST(α) .
- **FOLLOW(A)** is the set of the terminals $\overset{*}{\text{w}}$hich occur immediately after (follow) the *non-terminal A** in the strings derived from the starting symbol.
  - a terminal a is in FOLLOW(A) if S => αAaβ

Computing First

- To compute First(X) for all grammar symbols X, apply following rules until no more terminals or ε can be added to any First set:
  1. If X is a terminal then First(X) = {X}.
  2. If X is a nonterminal and X->Y1Y2…Yk is a production for some k>=1, then place a in First(X) if for some i a is in First(Yi) and ε is in all of First(Y1),…,First(Yi-1) that is Y1…Yi-1 => ε. if ε is in First(Yj) for j=1,…,k then add ε to First(X).
  3. If X-> ε is a production then add ε to First(X)

Computing follow

- To compute Follow(A) for all nonterminals A, apply following rules until nothing can be added to any follow set:
    1. Place $ in Follow(S) where S is the start symbol
    2. If there is a production A-> αBβ then everything in First(β) except ε is in Follow(B).
    3. If there is a production A->B or a production A->αBβ where First(β) contains ε, then everything in Follow(A) is in Follow(B)

Example

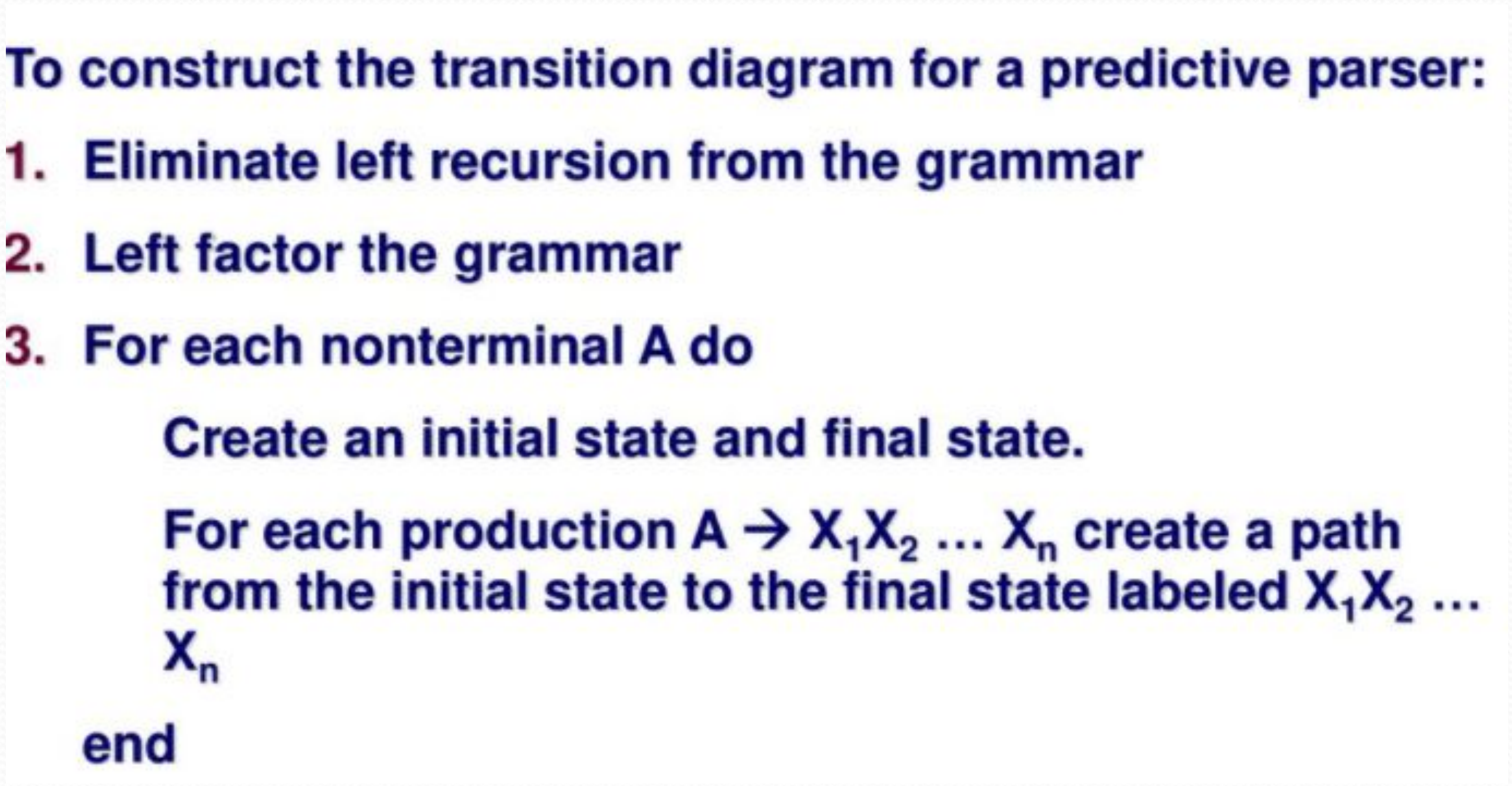
Consider the non-recursive grammar

E -> TE’	1). FIRST(E)= FIRST(T)=FIRST(F)={(,id}
E’ -> +TE’ | Ɛ	2). FIRST(E’)={+,Ɛ}
T -> FT’	3). FIRST (T’)= {*, Ɛ}
T’ -> *FT’ | Ɛ	4).FOLLOW(E)=FOLLOW(E’)={),$}
F -> (E) | **id**	5). FOLLOW(T)=FOLLOW(T’)={+,),$}
	6). FOLLOW(F)={+,*,),$}

LL(1) Grammars

- Predictive parsers are those recursive descent parsers needing no backtracking.
- Grammars for which we can create predictive parsers are called LL(1)
  - The first L means scanning input from left to right.
  - The second L means leftmost derivation.
  - And 1 stands for using one input symbol for look ahead.
- A grammar G is LL(1) if and only if whenever A-> α|βare two distinct productions of G, the following conditions hold:
  - For no terminal a do αandβ both derive strings beginning with a
  - At most one of α or βcan derive empty string
  - If $\beta \overset{*}{=>} \varepsilon$ then α does not derive any string beginning with a terminal in Follow(A).

Transition diagrams for Predictive Parser



To construct the transition diagram for a predictive parser:

1. **Eliminate left recursion from the grammar**
2. **Left factor the grammar**
3. **For each nonterminal A do**

   Create an initial state and final state.

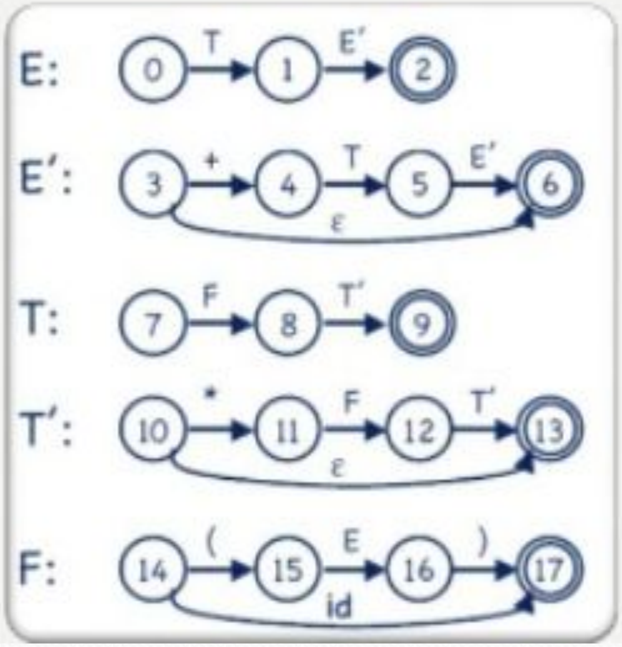
   For each production $A \rightarrow X_1X_2 \ldots X_n$ create a path from the initial state to the final state labeled $X_1X_2 \ldots X_n$

   end

Transition diagrams for Predictive Parser

- An expression grammar with left recursion and ambiguity removed:
- E -> T E'
- E' -> + T E' | ε
- T -> F T'
- T' -> * F T' | ε
- F -> (E) | id

- Corresponding transition diagrams



Transition diagrams for Predictive Parser

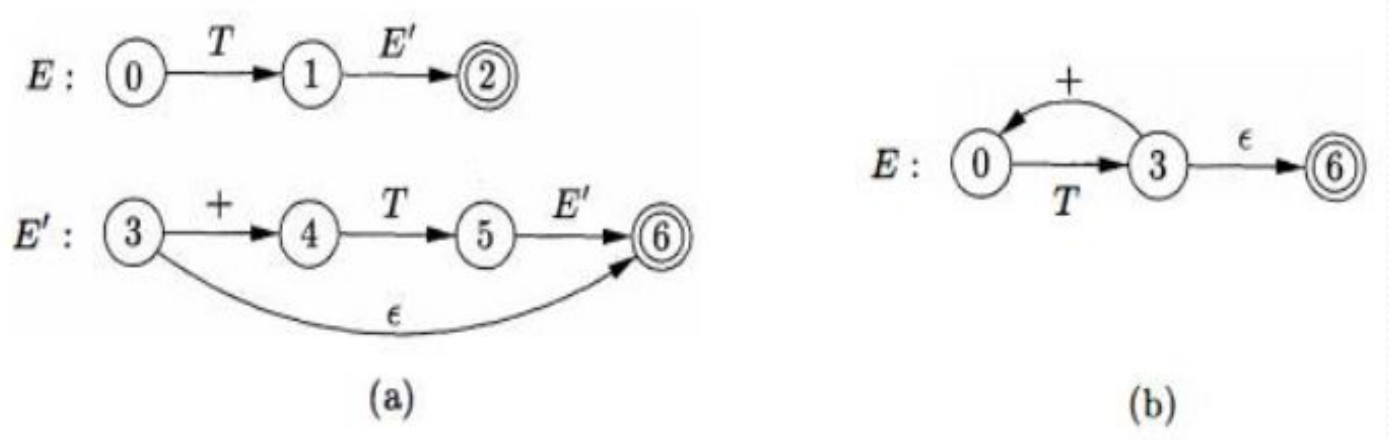


Figure 4.16: Transition diagrams for nonterminals E and E' of grammar 4.28

Construction of predictive parsing table

- INPUT: Grammar G
- OUTPUT: Parsing table M
- METHOD: For each production A->α in grammar do the following:
    1. For each terminal a in First(α) add A-> α in M[A,a]
    2. If ε is in First(α), then for each terminal b in Follow(A) add A-> ε to M[A,b].
    3. If ε is in First(α) and $ is in Follow(A), add A-> ε to M[A,$] as well

1. If after performing the above, there is no production in M[A,a] then set M[A,a] to error.

Example

E -> TE’
E’ -> +TE’ | Ɛ
T -> FT’
T’ -> *FT’ | Ɛ
F -> (E) | **id**

	First	Follow
F	{(,id}	{+, *,), $}
T	{(,id}	{+,), $}
E	{(,id}	{), $}
E’	{+,ε}	{), $}
T’	{*,ε}	{+,), $}

Non - terminal	Input Symbol					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E’ -> +TE’			E’ -> Ɛ	E’ -> Ɛ
T	T -> FT’			T -> FT’		
T’		T’ -> Ɛ	T’ -> *FT’		T’ -> Ɛ	T’ -> Ɛ
F	F -> **id**			F -> (E)		

Another example

S -> iEtSS' | a
S' -> eS | Ɛ
E -> b

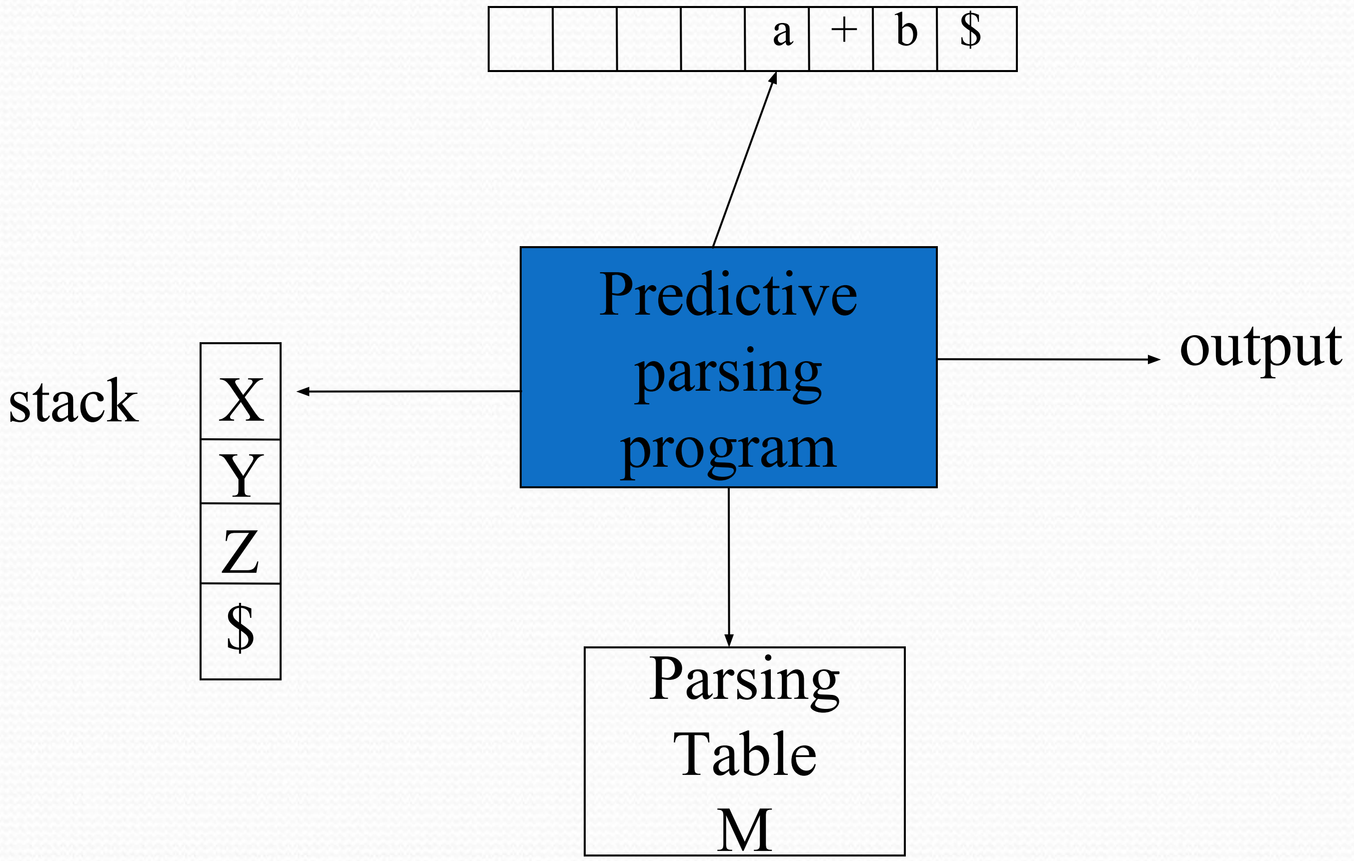
FIRST(S) = {i,a} FOLLOW(S) = {$,e}
FIRST(S') = {e,Ɛ} FOLLOW(S') = { $,e}
FIRST(E) = {b} FOLLOW(E) = { t }

Non - terminal	Input Symbol					
	a	b	e	i	t	$
S	S -> a			S -> iEtSS'		
S'			S' -> Ɛ S' -> eS			S' -> Ɛ
E		E -> b				

Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.
- In LL(1)
  - the first “L“ $\rightarrow$ scanning the input from left to right and
  - second ”L” $\rightarrow$ producing a leftmost derivation and
  - the “1” $\rightarrow$ one input symbol of look ahead at each step
- It uses stack explicitly.
- In non recursive predictive parser ,production is applied on the parsing table

Non-recursive predicting parsing



LL(1)Parser

Input buffer

- Input string to be parsed .The end of the string is marked with a special symbol $.

Output

- A production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

Stack

- Contains the grammar symbols
- At the bottom of the stack, there is a special end marker symbol $.
- Initially the stack contains only the symbol $ and the starting symbol S.

  i.e; $S **<=** initial stack
- When the stack is emptied (ie. only $ left in the stack), the parsing is completed.

Parsing table

- A two-dimensional array M[A,a]
- Each row (A) ,is a non-terminal symbol
- Each column (a), is a terminal symbol or the special symbol $

LL(1)Parser –Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are FOUR possible PARSER ACTIONS:-
- If X = a = $ **->** parser halts and announces successful completion of the parsing
- If X = a <> $ **->** parser pops X from the stack, and advances the input pointer to the next input symbol
- If X is a non-terminal

  -> parser looks at the parsing table entry M[X,a]. If M[X,a] holds a production rule X**->**Y1Y2...Yk, it pops X from the stack and pushes Yk,Yk-1,...,Y1 into the stack. The parser also outputs the production rule X**->**Y1Y2...Yk to represent a step of the derivation.
- none of the above **->** error
  - all empty entries in the parsing table are errors.
  - If X is a terminal symbol different from a, this is also an error case.

Non Recursive Predictive Parsing program

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

Output : If w is in L(G), a leftmost derivation of w ;

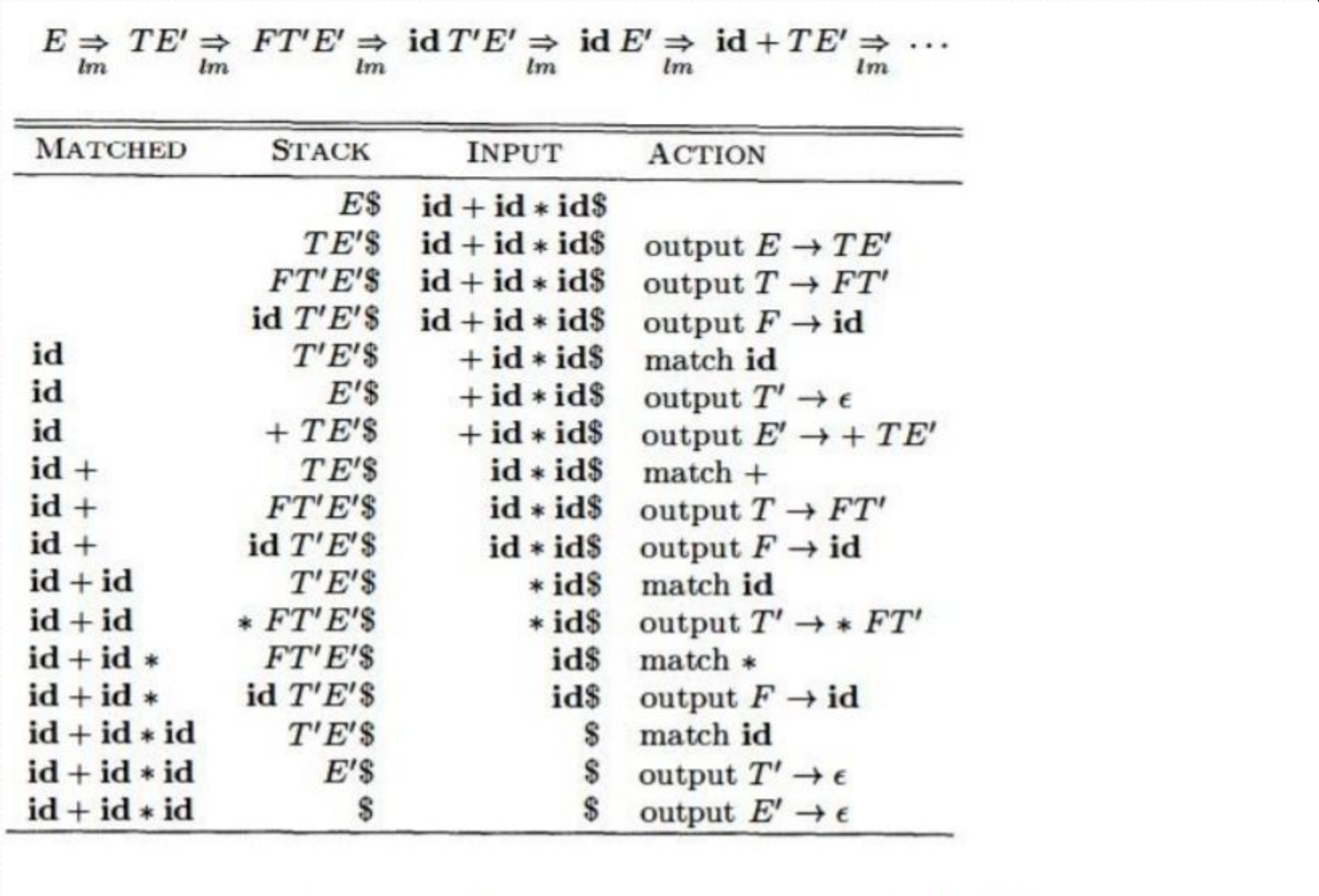
Otherwise, an error indication

Method : Initially parser is in configuration ,it has $S on the stack with S , the start symbol of G on top ,and w$ in the input buffer. The program that utilizes the parsing table M to produce a parse for the input

Algorithm:

Predictive parsing algorithm

```
Algorithm:
Let a be the first symbol of w;
Let X to the top stack symbol;
While (X<>$) { /* stack is not empty */
   if (X is a) pop the stack and advance ip;
   else if (X is a terminal) error();
   else if (M[X,a] is an error entry) error();
   else if (M[X,a] = X->Y1Y2..Yk) {
      output the production X->Y1Y2..Yk;
      pop the stack;
      push Yk,…,Y2,Y1 on to the stack with Y1 on top;
   }
   set X to the top stack symbol;
}
```

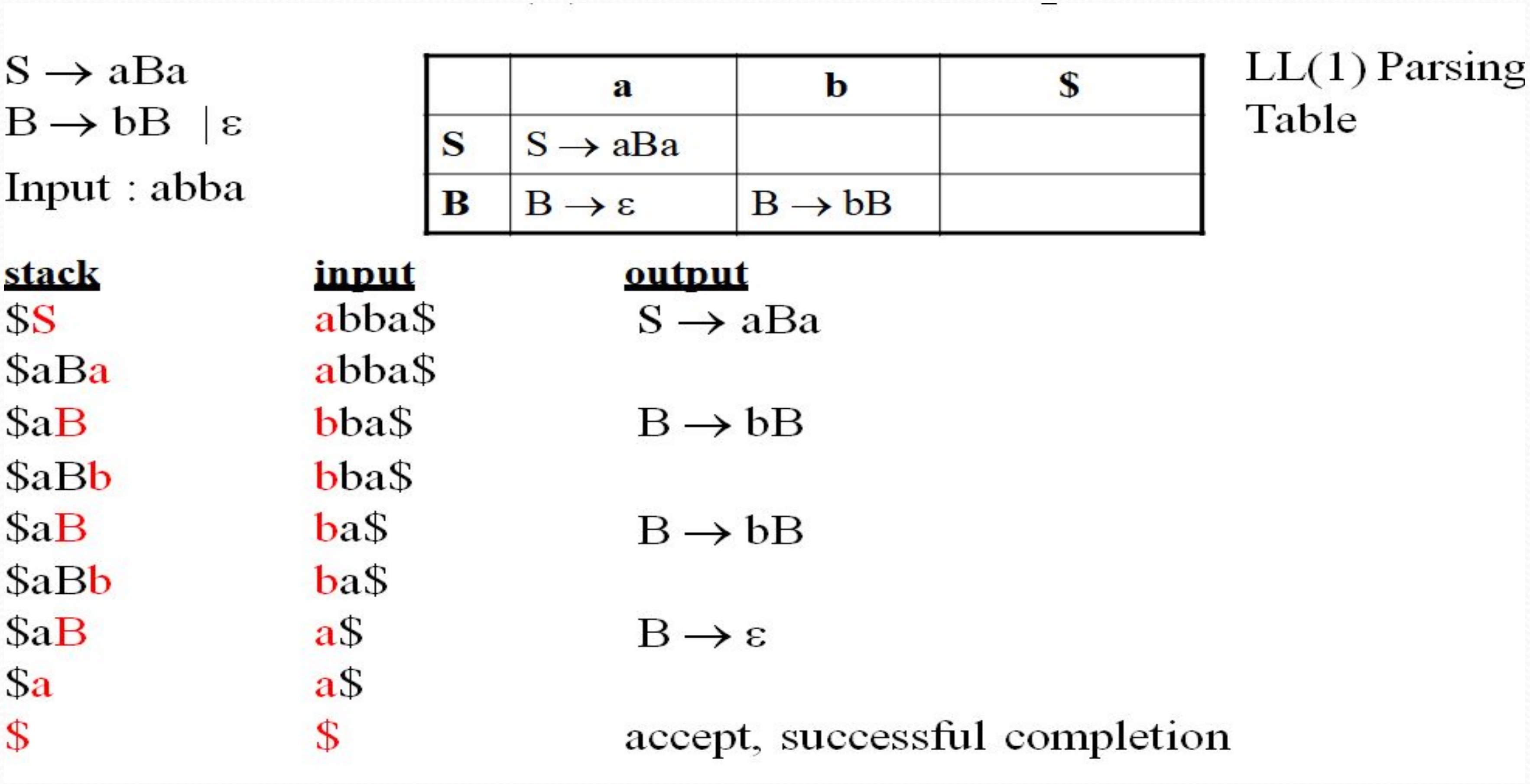


$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} \mathbf{id}\,T'E' \underset{lm}{\Rightarrow} \mathbf{id}\,E' \underset{lm}{\Rightarrow} \mathbf{id} + TE' \underset{lm}{\Rightarrow} \cdots$$

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Figure 3: Moves made by predictive parser on input id+id*id

LL(1) Parser Example



$S \rightarrow aBa$
$B \rightarrow bB \mid \varepsilon$
Input : abba

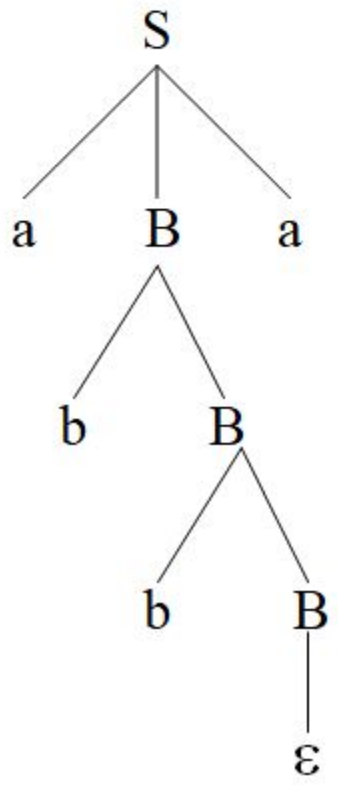
	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

LL(1) Parsing Table

stack	input	output
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	bba$	$B \rightarrow bB$
$aBb	bba$	
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	accept, successful completion

Outputs: $S \to aBa$ $B \to bB$ $B \to bB$ $B \to \varepsilon$

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



parse tree

LL(1) Parser Example

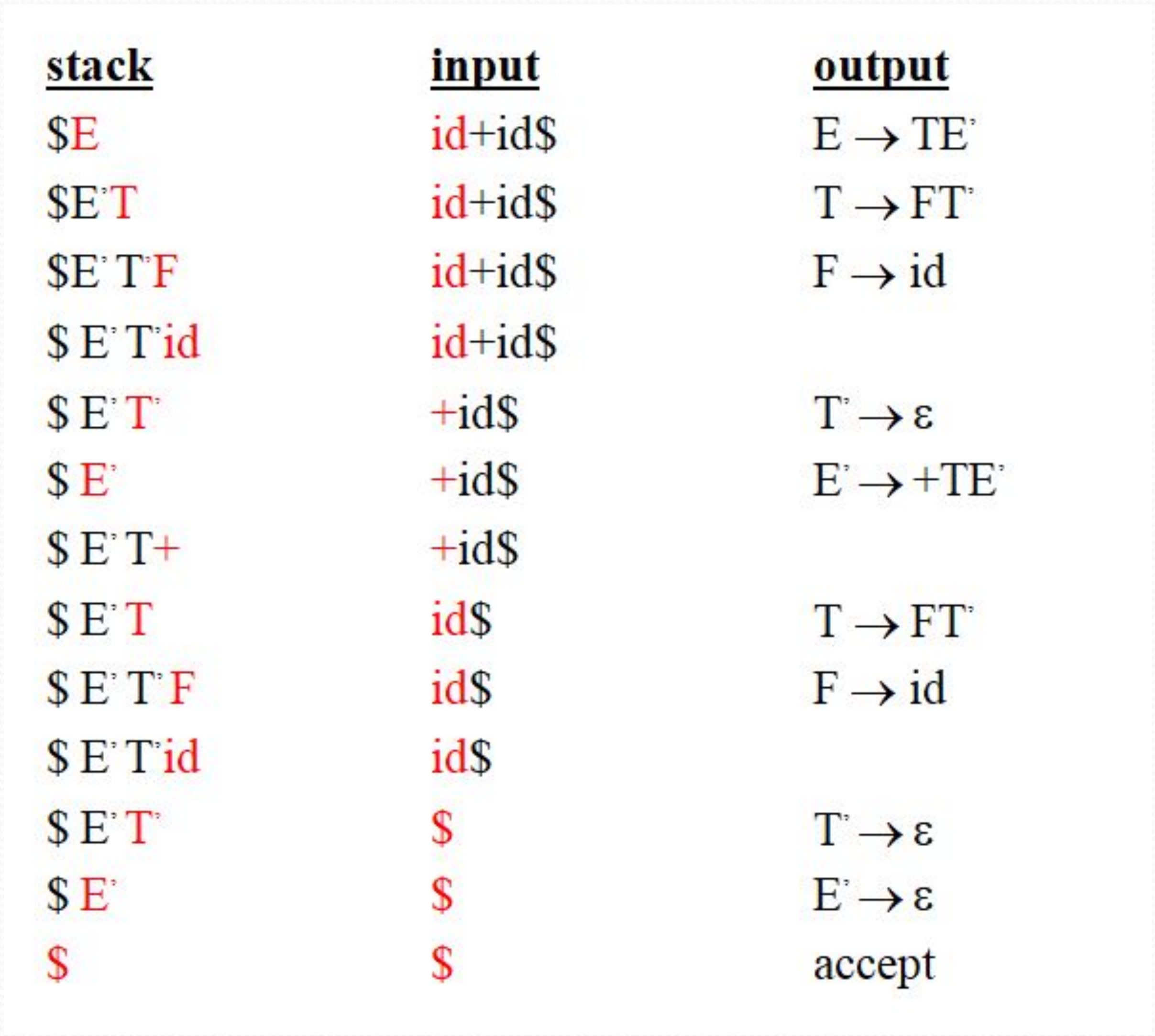
$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow id \mid (E)$

$\longrightarrow$

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

Input : id +id

	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	$F \rightarrow id$			$F \rightarrow (E)$		



stack	**input**	**output**
$\$E$	id+id$	$E \rightarrow TE'$
$\$E'T$	id+id$	$T \rightarrow FT'$
$\$E'T'F$	id+id$	$F \rightarrow id$
$\$E'T'id$	id+id$	
$\$E'T'$	+id$	$T' \rightarrow \varepsilon$
$\$E'$	+id$	$E' \rightarrow +TE'$
$\$E'T+$	+id$	
$\$E'T$	id$	$T \rightarrow FT'$
$\$E'T'F$	id$	$F \rightarrow id$
$\$E'T'id$	id$	
$\$E'T'$	$	$T' \rightarrow \varepsilon$
$\$E'$	$	$E' \rightarrow \varepsilon$
$	$	accept

Error recovery in predictive parsing

- An error may occur in the predictive parsing (LL(1) parsing)
    - if the terminal symbol on the top of stack does not match with the current input symbol.
    - if the top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A,a] is empty.
- What should the parser do in an error case?
    - The parser should be able to give an error message (as much as possible meaningful error message).
    - It should be recover from that error case, and it should be able to continue the parsing with the rest of the input.

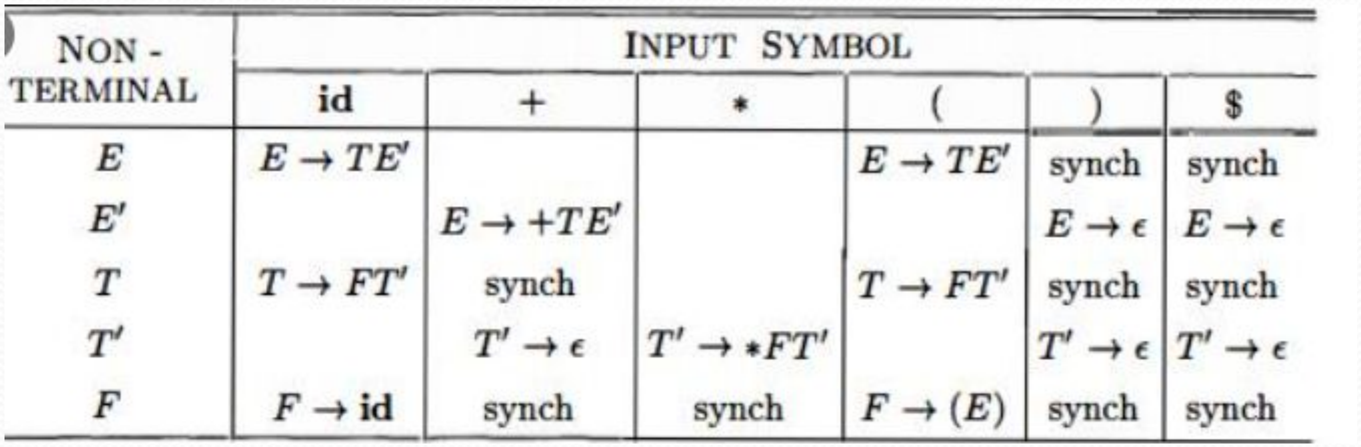
Error recovery in predictive parsing

- Panic mode
  - Place all symbols in Follow(A) into synchronization set for nonterminal A: skip tokens until an element of Follow(A) is seen and pop A from stack.
  - Add to the synchronization set of lower level construct the symbols that begin higher level constructs
  - Add symbols in First(A) to the synchronization set of nonterminal A
  - If a nonterminal can generate the empty string then the production deriving can be used as a default
  - If a terminal on top of the stack cannot be matched, pop the terminal, issue a message saying that the terminal was insterted

Error recovery in predictive parsing

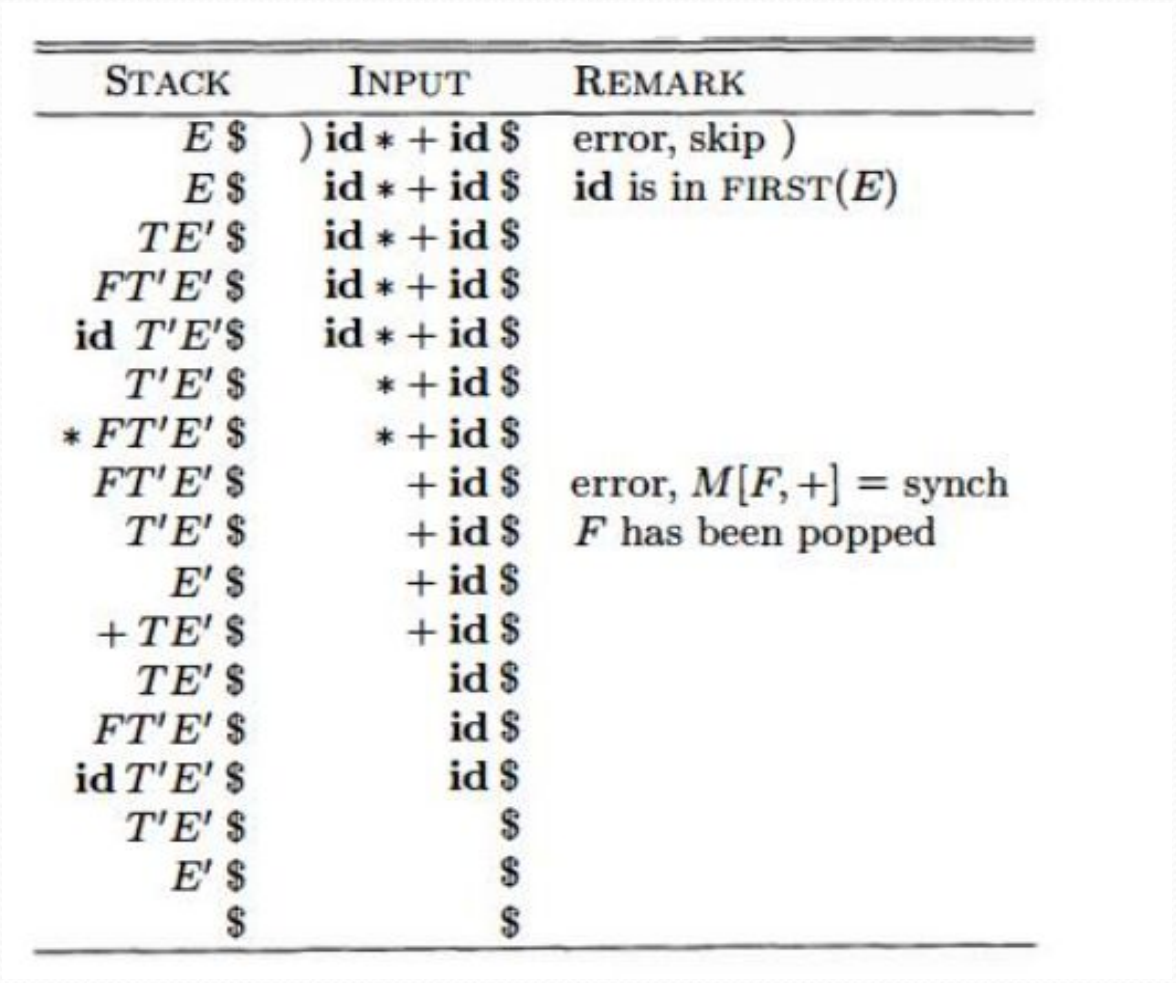
- In ***panic-mode error*** recovery, we skip all the input symbols until a synchronizing token is found.
- All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token (“synch “) for that non-terminal.
- “ synch “ is placed in the parsing table for the positions of follow set of that non terminal.
- If the parser looks up entry “ M [A ,a] “ and finds that it is blank ,then the input symbol a is skipped
- If the entry is “synch “ then the non terminal on top of the stack is popped in an attempt to resume parsing
- If the token on top of the stack does not match the input symbol ,then we pop the token from the stack.

Synchronizing tokens added to the parsing table



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

Parsing and error recovery moves made by a predictive parser



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

Phrase-Level Error Recovery

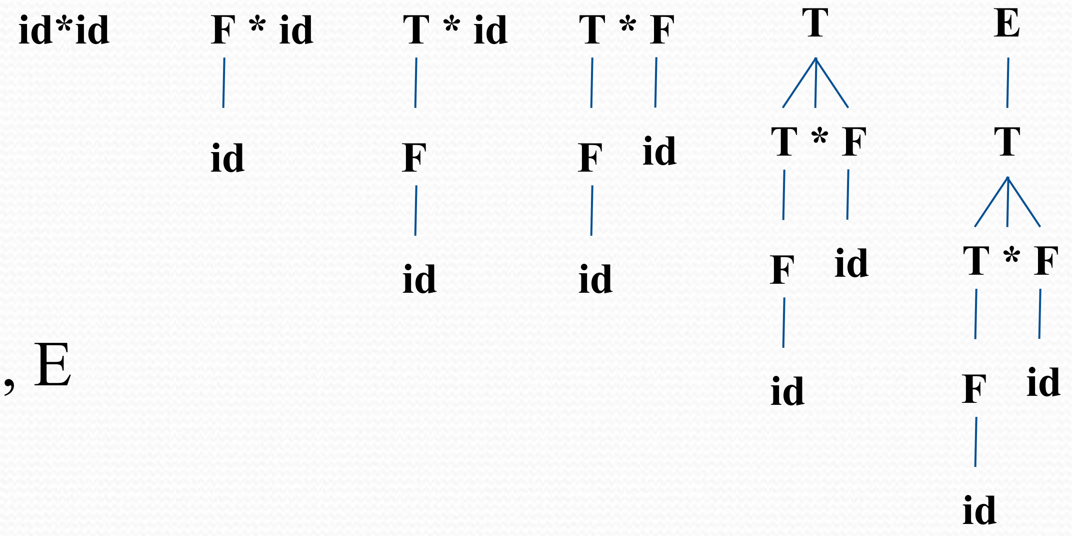
- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
  - change, insert, or delete input symbols.
  - issue appropriate error messages
  - pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.

Bottom-up Parsing

Introduction

- Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)
- Example: id*id

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



id*id, F*id, T*id, T*F, T, E

Shift-reduce parser

- The general idea is to shift some symbols of input to the stack until a reduction can be applied
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply
- A reduction is a reverse of a step in a derivation
- The goal of a bottom-up parser is to construct a derivation in reverse:
  - E=>T=>T*F=>T*id=>F*id=>id*id

Handle pruning

- A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation

Right sentential form	Handle	Reducing production
id_1*id_2	id_1	F->id
$F*id_2$	F	T->
$T*id_2$	id_2	F->id
T*F	T*F	E->T*F

- A **handle** of a right sentential form $\gamma (\equiv \alpha\beta\omega)$ is

  a production rule $A \rightarrow \beta$ and a position of γ

  where the string β may be found and replaced by A to produce

  the previous right-sentential form in a rightmost derivation of γ.

$$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A \omega \underset{rm}{\Rightarrow} \alpha\beta\omega$$

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- We will see that ω is a string of terminals.

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$$S=\gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \cdots \Rightarrow_{rm} \gamma_{n-1} \Rightarrow_{rm} \gamma_n = \omega$$

input string

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Right-Most Derivation of `id+id*id`

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id \Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$

Right Sentential Form	Handle	Reducing Production
$\underline{id}+id*id$	id	$F \rightarrow id$
$\underline{F}+id*id$	F	$T \rightarrow F$
$\underline{T}+id*id$	T	$E \rightarrow T$
$E+\underline{id}*id$	id	$F \rightarrow id$
$E+\underline{F}*id$	F	$T \rightarrow F$
$E+T*\underline{id}$	id	$F \rightarrow id$
$E+\underline{T*F}$	T*F	$T \rightarrow T*F$
$\underline{E+T}$	E+T	$E \rightarrow E+T$
E		

Handles are red and underlined in the right-sentential forms

Shift reduce parsing

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

Stack	Input
$	w$

- Acceptance configuration

Stack	Input
$S	$

Shift reduce parsing (cont.)

- There are four possible actions of a shift-parser action:

  1. **Shift** : The next input symbol is shifted onto the top of the stack.
  2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
  3. **Accept:** Successful completion of parsing .
  4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.

- Initial stack just contains only the end-marker $.

- The end of the input string is marked by the end-marker $.

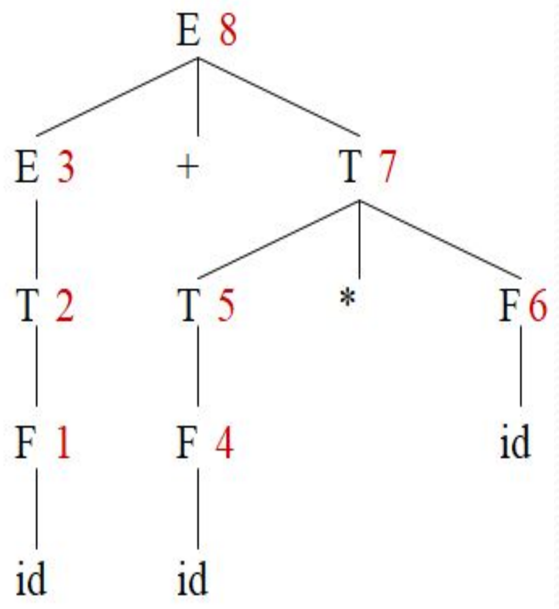
Shift reduce parsing (cont.)

- Basic operations:
  - Shift
  - Reduce
  - Accept
  - Error
- Example: id*id

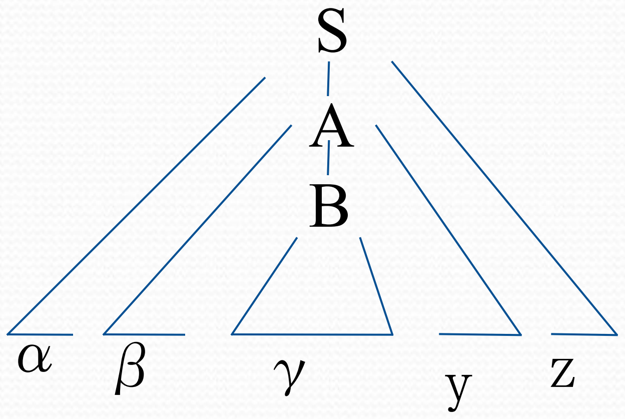
Stack	Input	Action
$	id*id$	shift
$id	*id$	reduce by F->id
$F	*id$	reduce by T->F
$T	*id$	shift
$T*	id$	shift
$T*id	$	reduce by F->id
$T*F	$	reduce by T->T*F
$T	$	reduce by E->T
$E	$	accept

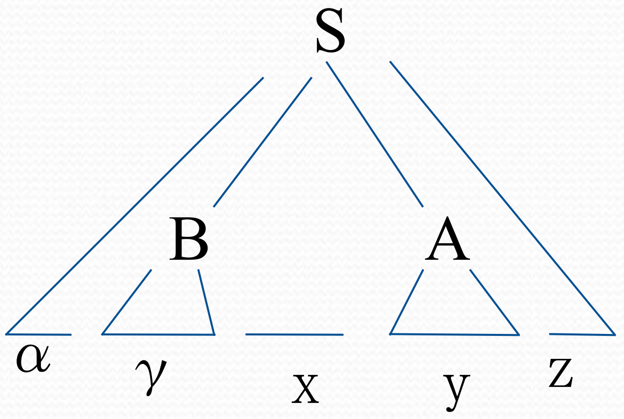
Stack	**Input**	**Action**
$	id+id*id$	shift
$id	+id*id$	reduce by F → id
$F	+id*id$	reduce by T → F
$T	+id*id$	reduce by E → T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by F → id
$E+F	*id$	reduce by T → F
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	reduce by F → id
$E+T*F	$	reduce by T → T*F
$E+T	$	reduce by E → E+T
$E	$	accept

Parse Tree



Handle will appear on top of the stack





Stack	Input
$\$\ \alpha\ \beta\ \gamma$	yz$
$\$\ \alpha\ \beta\ B$	yz$
$\$\ \alpha\ \beta\ By$	z$

Stack	Input
$\$\ \alpha\ \gamma$	xyz$
$\$\ \alpha\ Bxy$	z$

Conflicts during shift reduce parsing

- Two kind of conflicts
  - Shift/reduce conflict
  - Reduce/reduce conflict
- Example:

stmt $\longrightarrow$ **If** expr **then** stmt
| **If** expr **then** stmt **else** stmt
| **other**

Stack	Input
… if expr then stmt	else …$

Reduce/reduce conflict

1) stmt -> id(parameter_list)
2) stmt -> expr:=expr
3) parameter_list->parameter_list, parameter
4) parameter_list->parameter
5) parameter->id
6) expr->id(expr_list)
7) expr->id
8) expr_list->expr_list, expr
9) expr_list->expr

Stack	Input
… id(id	,id) …$