CS6109 – COMPILER DESIGN

Module – 5

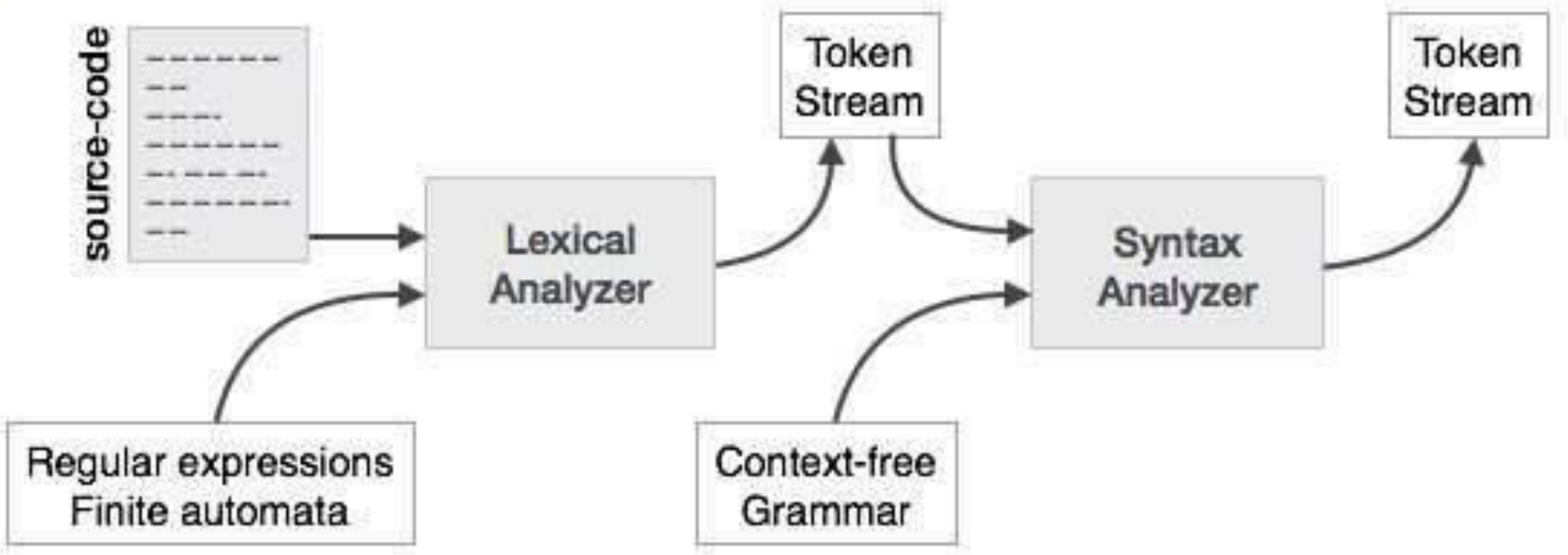
Presented By
Dr. S. Muthurajkumar,
Assistant Professor,
Dept. of CT, MIT Campus,
Anna University, Chennai.

MODULE - 5

- Recursive Descent Parsers
- LL(1) Parsers
- Shift Reduce Parser
- LR(0) items
- Simple LR parser

SYNTAX ANALYZERS

- A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams.
- The parser analyzes the source code (token stream) against the production rules to detect any errors in the code.
- The output of this phase is a parse tree.



ELIMINATING AMBIGUITY

- Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.
- As an example, we shall eliminate the ambiguity from the following "dangling else" grammar:

  stmt → if expr then stmt

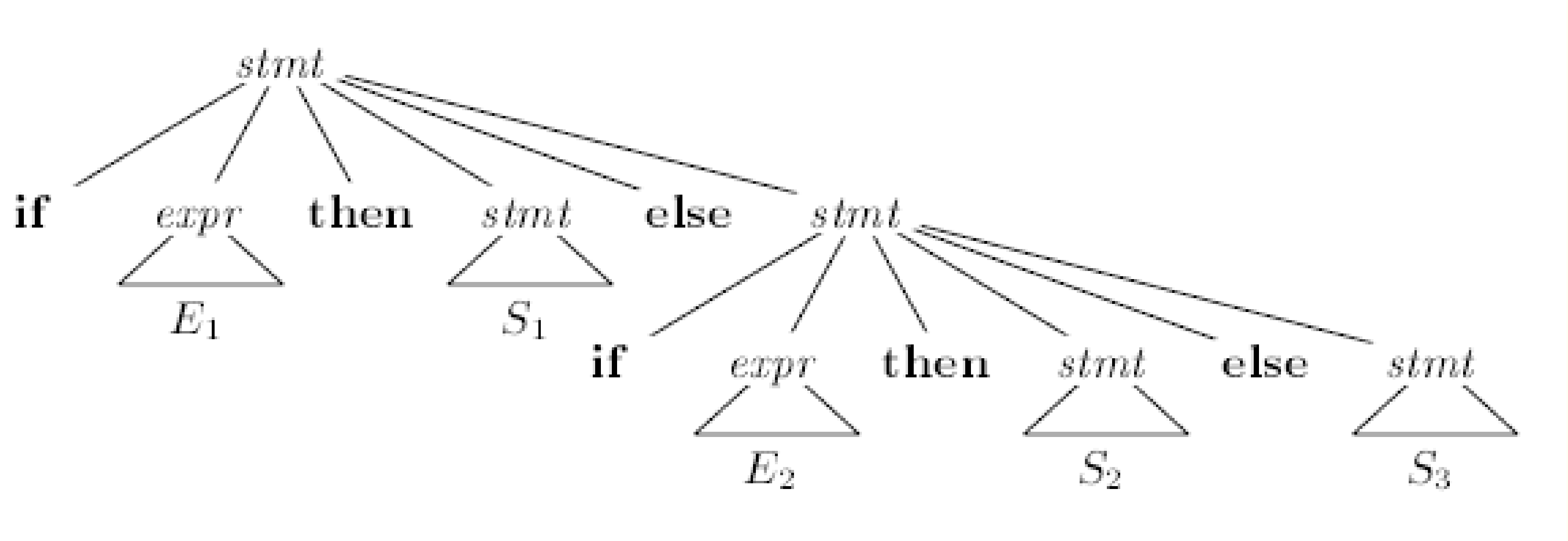
  → if expr then stmt else stmt

  → other

- Here "other" stands for any other statement. According to this grammar, the compound conditional statement

  if E_1 then S_1 else if E_2 then S_2 else S_3

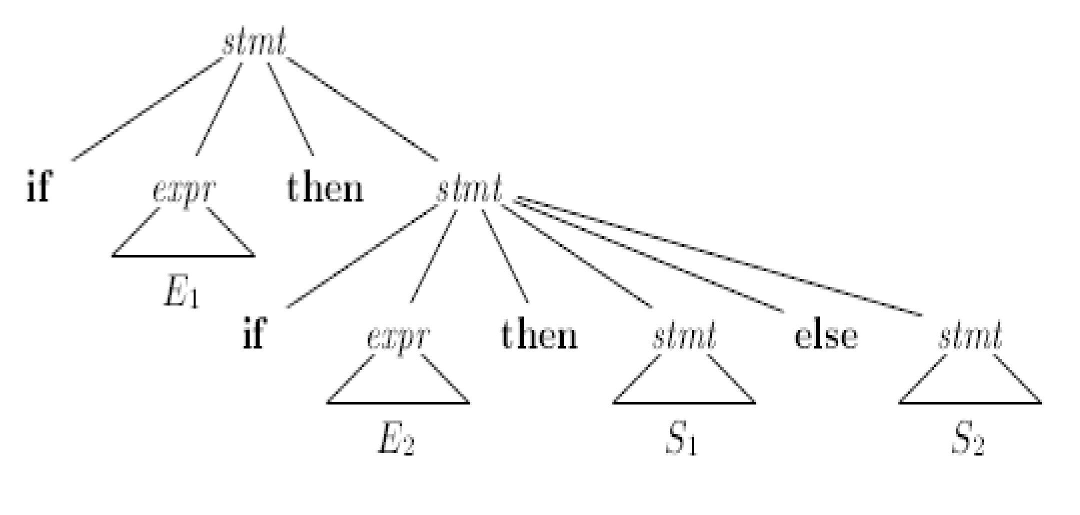
ELIMINATING AMBIGUITY

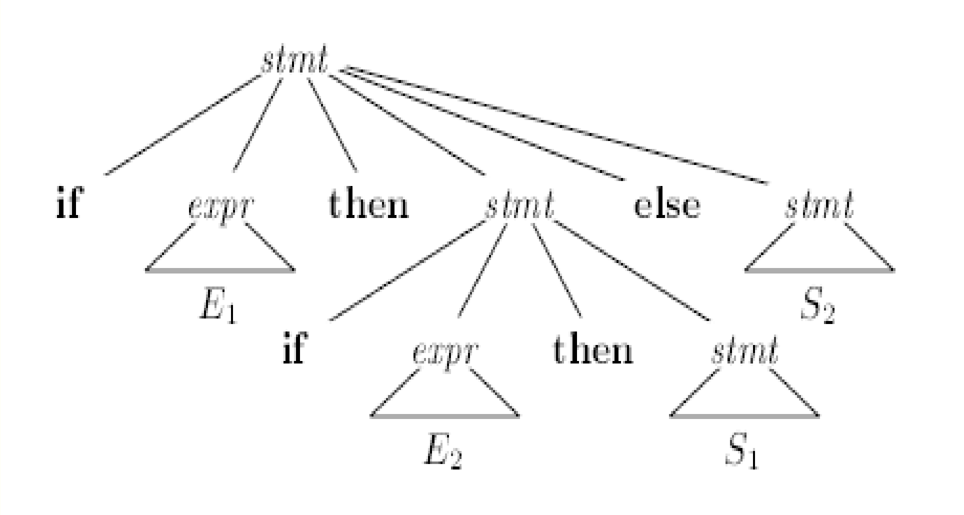


Parse tree for a conditional statement

ELIMINATING AMBIGUITY

- if E_1 then if E_2 then S_1 else S_2
- Two parse trees for an ambiguous sentence





ELIMINATING AMBIGUITY

- **Unambiguous grammar for if-then-else statements**

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 nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.
- Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.
- Left-recursive pair of productions

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

- Eliminate left recursion $A' \rightarrow \varepsilon$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

ELIMINATION OF LEFT RECURSION

E → E + T | T

T → T * F | F

F → (E) | id

- Left-recursive pair of productions

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

- **Eliminate left recursion**

  $A \rightarrow \beta A'$

  $A' \rightarrow \alpha A' \mid \epsilon$

- **After Eliminate left recursion** E → E + T | T

  E → T E'

  E' → + T E' | ϵ

ELIMINATION OF LEFT RECURSION

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

- Left-recursive pair of productions

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

- **Eliminate left recursion**

  $A \rightarrow \beta A'$

  $A' \rightarrow \alpha A' \mid \epsilon$

- **After Eliminate left recursion** $T \rightarrow T * F \mid F$

$T \rightarrow F\ T'$

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

ELIMINATION OF LEFT RECURSION

- S → A a | b
- A → A c | S d | ϵ
- The nonterminal S is left recursive because S => Aa => Sda, but it is not immediately left recursive.

LEFT FACTORING

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing.
- When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.
- For example, if we have the two productions

  stmt → if expr then stmt else stmt

  → if expr then stmt
- on seeing the input if, we cannot immediately tell which production to choose to expand stmt.
- $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

LEFT FACTORING

- Before Left Factoring

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

- After Left Factoring

  $A \rightarrow \alpha A'$

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

- **Example**

  $S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a$

  $E \rightarrow b$

- After Left Factoring

  $S \rightarrow i\ E\ t\ S\ S' \mid a$

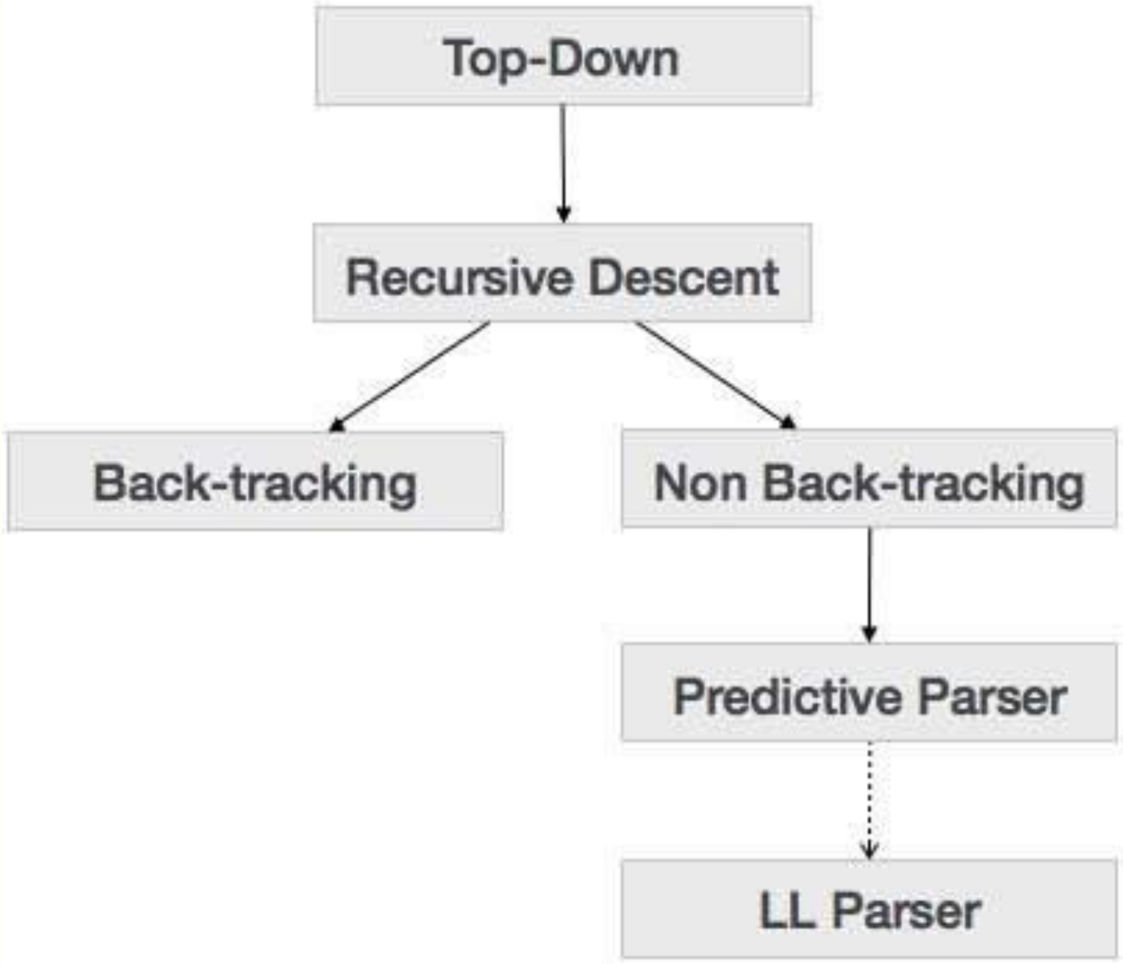
  $S' \rightarrow e\ S \mid \epsilon$

  $E \rightarrow b$

RECURSIVE DESCENT PARSERS

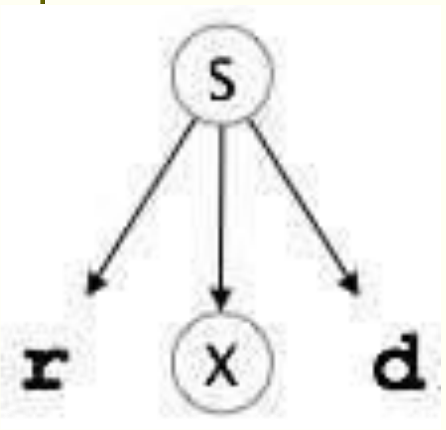
- Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right.
- It uses procedures for every terminal and non-terminal entity.
- This parsing technique recursively parses the input to make a parse tree, which may or may not require backtracking.
- But the grammar associated with it (if not left factored) cannot avoid backtracking.
- 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.

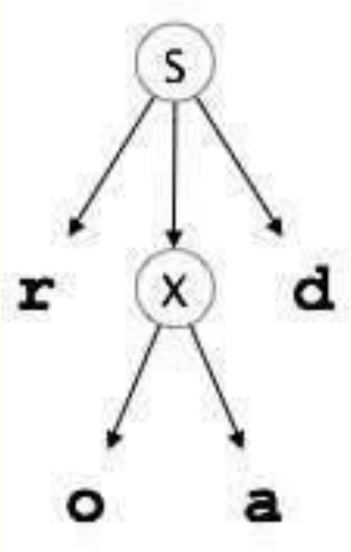
RECURSIVE DESCENT PARSERS

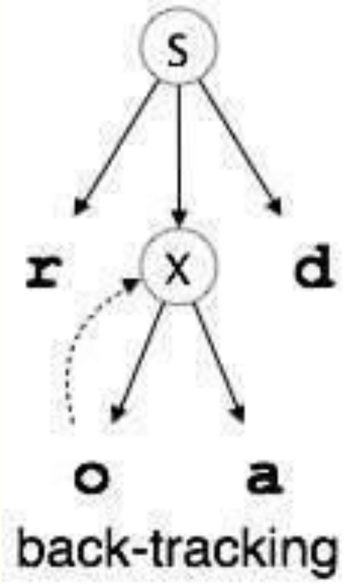


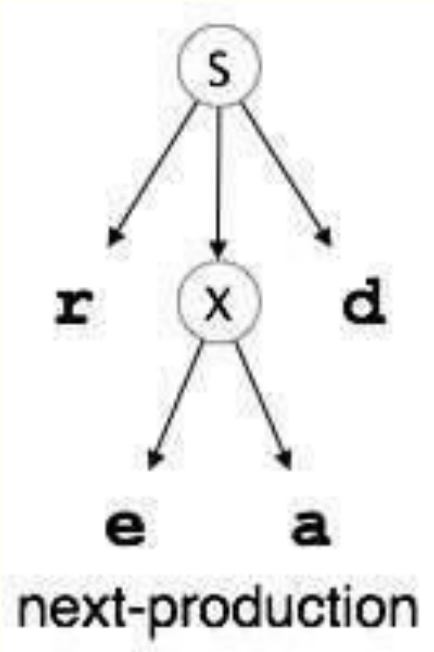
BACKTRACKING

- Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched).
- $S \rightarrow rXd \mid rZd$
- $X \rightarrow oa \mid ea$
- $Z \rightarrow ai$









- Now the parser matches all the input letters in an ordered manner. The string is accepted.

BACKTRACKING

- 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.

TOP-DOWN PARSING

- Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder.
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

  $E \rightarrow T\ E'$

  $E' \rightarrow +\ T\ E' \mid \epsilon$

  $T \rightarrow F\ T'$

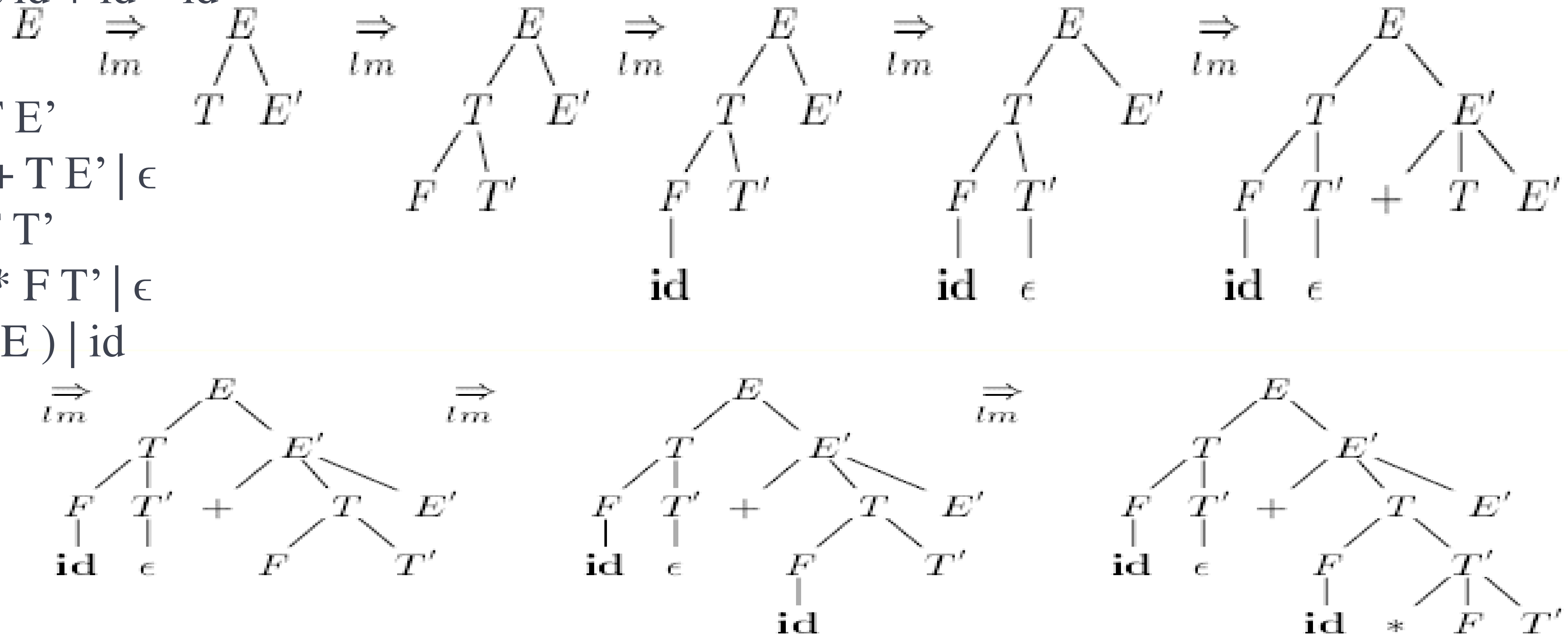
  $T' \rightarrow *\ F\ T' \mid \epsilon$

  $F \rightarrow (\ E\ ) \mid id$
- Input id + id * id

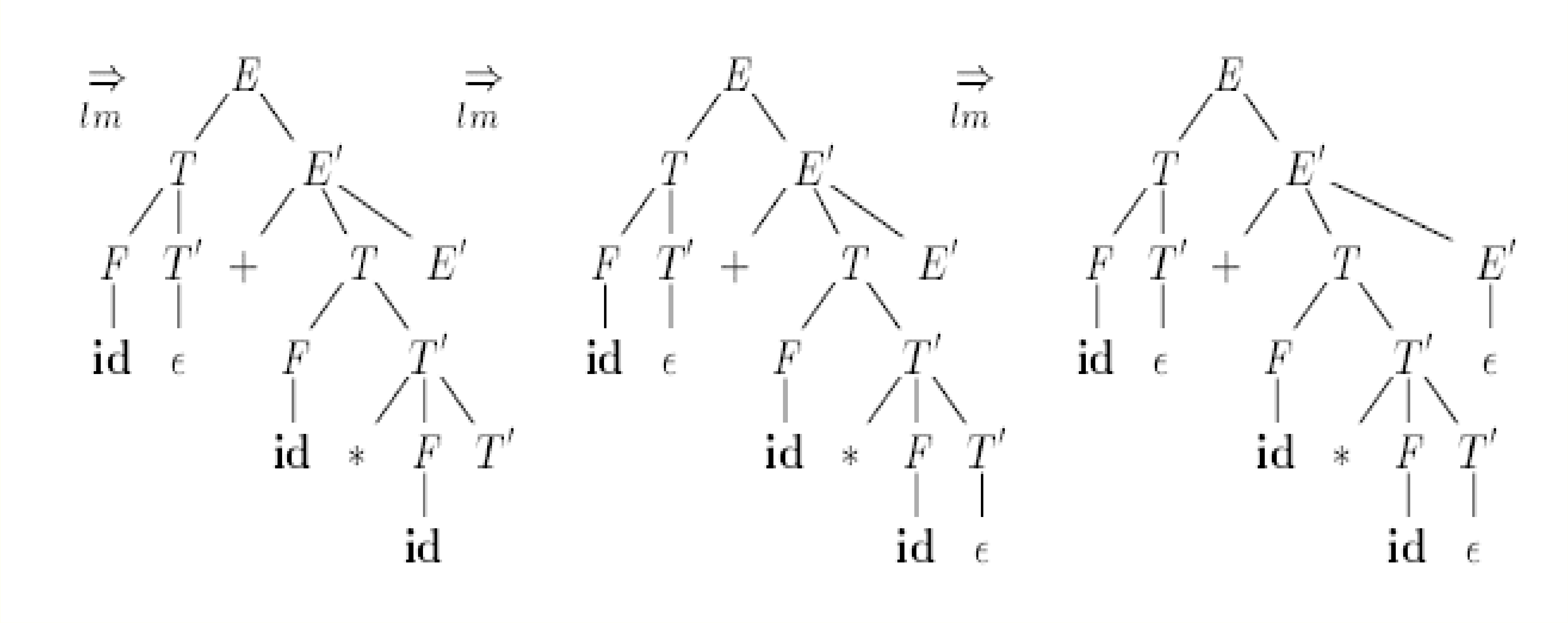
TOP-DOWN PARSING

Input id + id * id

E → T E’
E’ → + T E’ | ϵ
T → F T’
T’ → * F T’ | ϵ
F → (E) | id



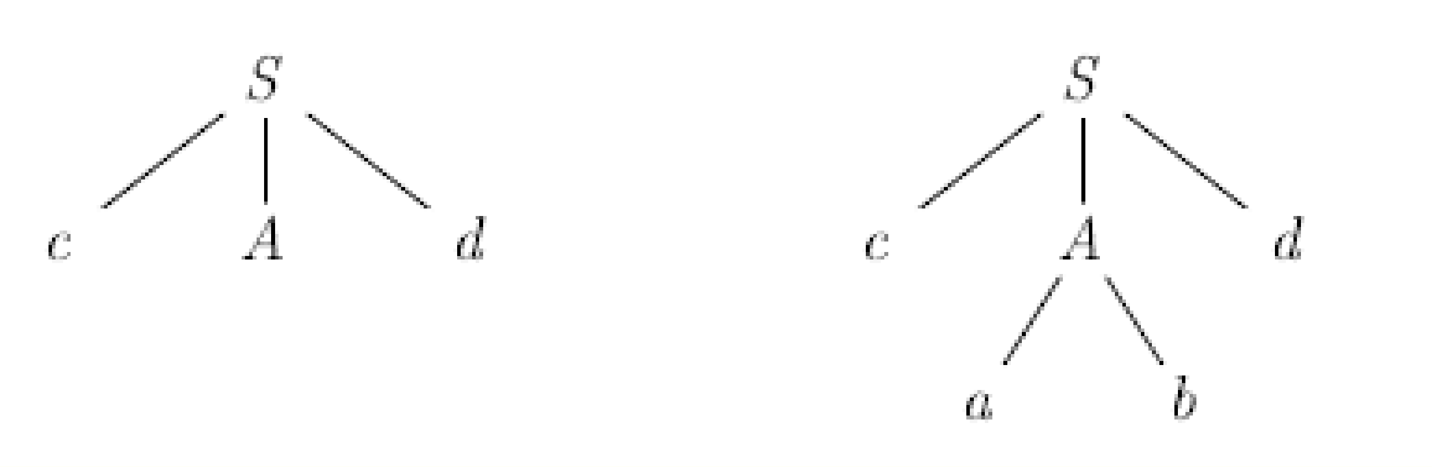
TOP-DOWN PARSING



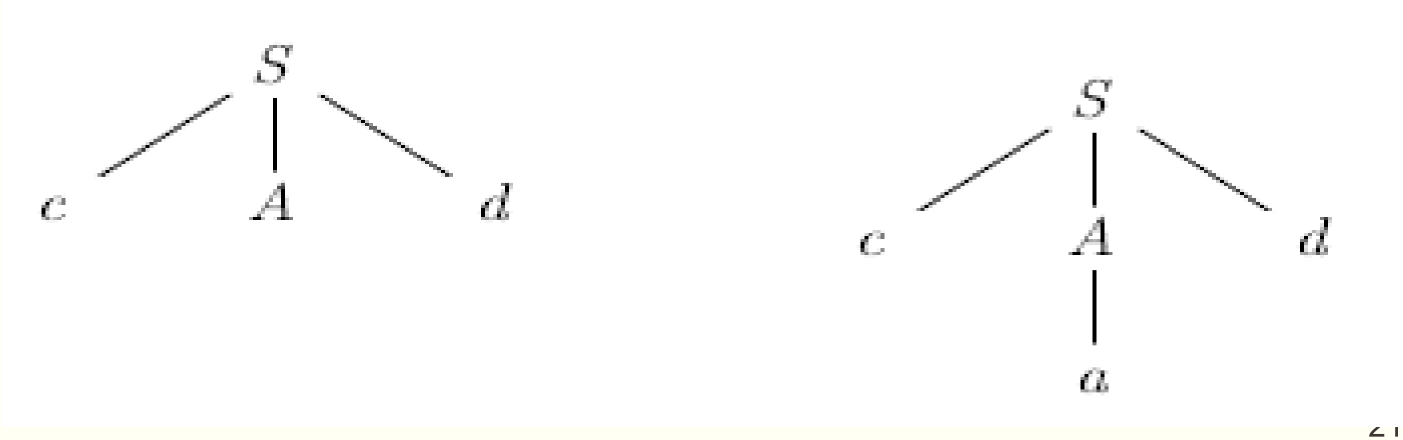
Top-down parse for id + id * id

TOP-DOWN PARSING

- Consider the grammer
- S → c A d
- A → a b | a
- Input string w = cad



- Not accepted so backtracking



- Now accepted

FIRST and FOLLOW

1. If X is a terminal, then FIRST(X) = {X}.
2. If X is a nonterminal and $X \rightarrow Y_1Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), … ,FIRST(Y_{i-1}); that is, $Y_1 \ldots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all j = 1, 2, … ,k, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST(Y2), and so on.
3. 3. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

FIRST and FOLLOW

- **Example**

  E → T E'

  E' → + T E' | ϵ

  T → F T'

  T' → * F T' | ϵ

  F → (E) | id
- First (E) = {(, id}
- First (E') = {+, ϵ}
- First (T) = {(, id}
- First (T') = {*, ϵ}
- First (F) = {(, id}

FIRST and FOLLOW

- **Example**

  S → i E t S S' | a

  S' → e S| ϵ

  E → b
- First (S) = {i, a}
- First (S') = {e, ϵ}
- First (E) = {b}

FIRST and FOLLOW

1. Place $ in FOLLOW(S), where S is the start symbol, and $ is the input right endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

FIRST and FOLLOW

- **Example**

  $E \rightarrow T\ E'$

  $E' \rightarrow +\ T\ E' \mid \epsilon$

  $T \rightarrow F\ T'$

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

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

- First (E) = {(, id} — Follow (E) = {), $}
- First (E') = {+, ϵ} — Follow (E') = {), $}
- First (T) = {(, id} — Follow (T) = {+,), $}
- First (T') = {*, ϵ} — Follow (T') = {+,), $}
- First (F) = {(, id} — Follow (F) = {*, +,), $}

FIRST and FOLLOW

- **Example**

  $S \rightarrow i\ E\ t\ S\ S' \mid a$

  $S' \rightarrow e\ S \mid \epsilon$

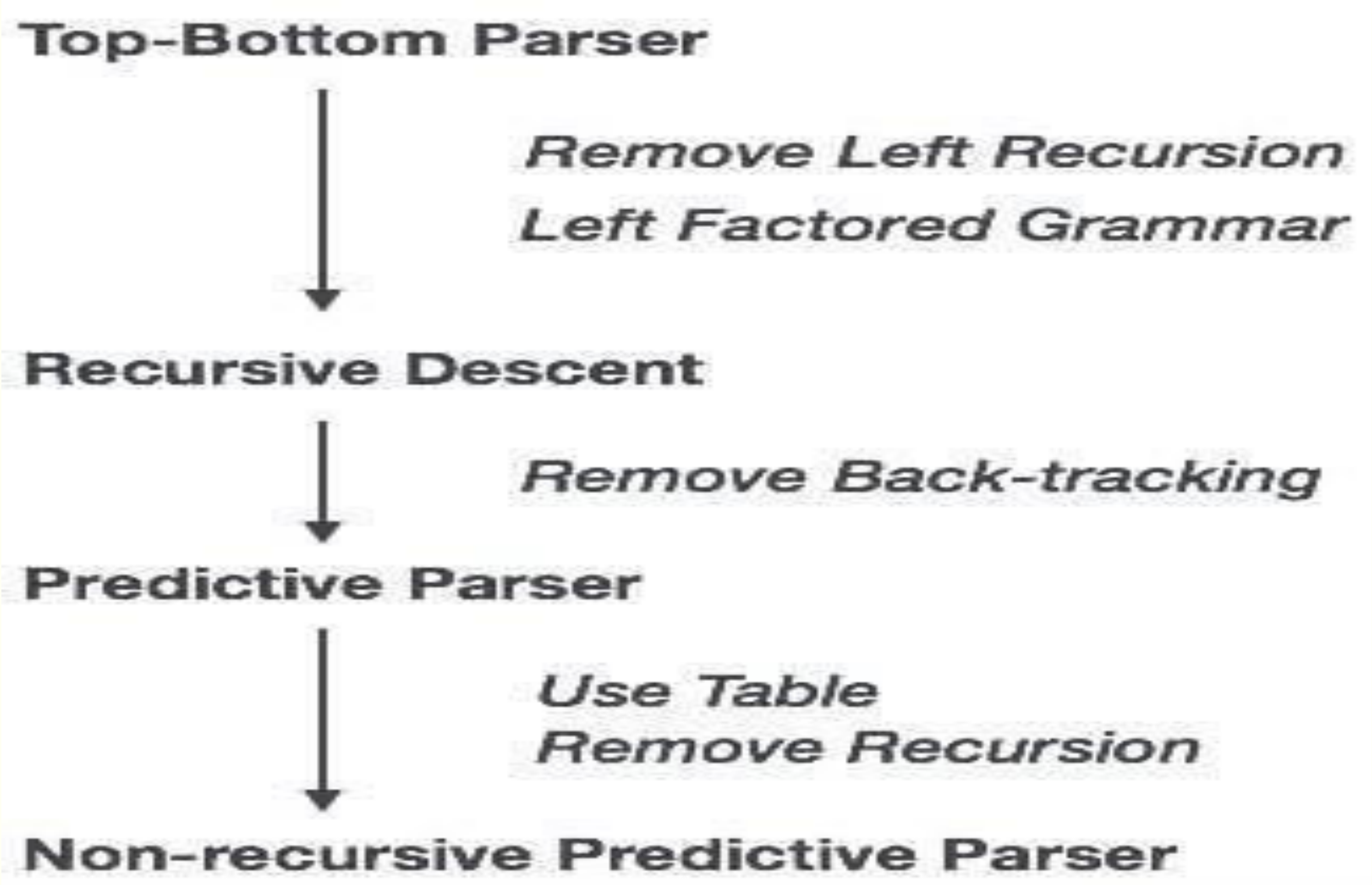
  $E \rightarrow b$

- First (S) = {i, a}
- First (S') = {e, ϵ}
- First (E) = {b}
- Follow (S) = {e, \$}
- Follow (S') = {e, \$}
- Follow (E) = {t}

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.

PREDICTIVE PARSER



PREDICTIVE PARSER

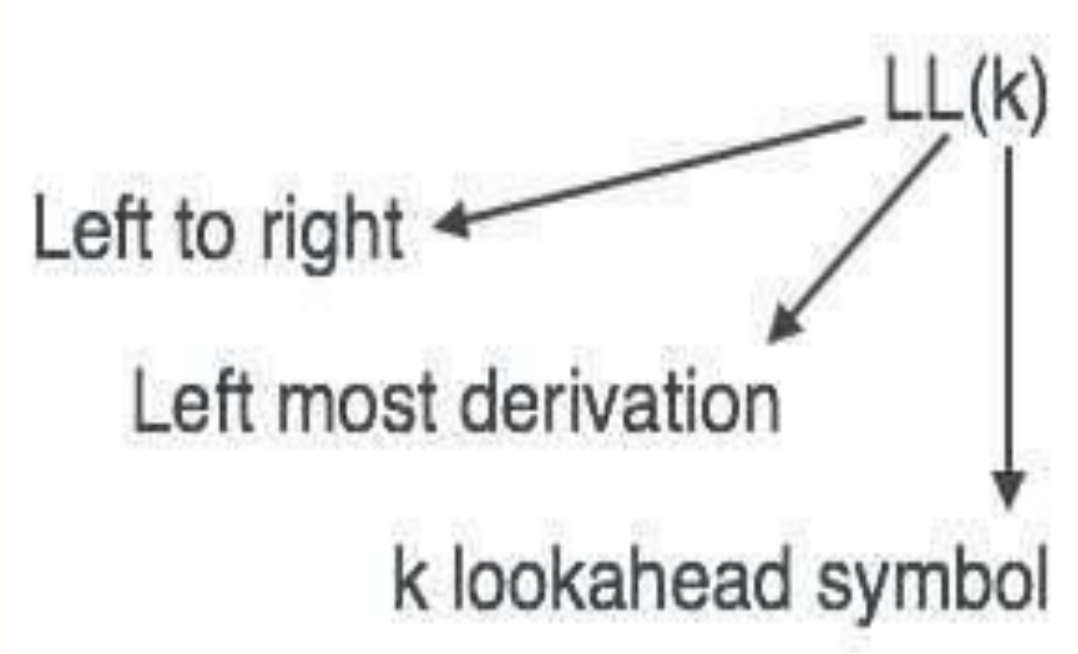
- Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree.
- Both the stack and the input contains an end symbol $ to denote that the stack is empty and the input is consumed.
- The parser refers to the parsing table to take any decision on the input and stack 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 (1) PARSER

- 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 lookaheads.
- Generally k = 1, so LL(k) may also be written as LL(1).



PREDICTIVE PARSER

- Construction of Predictive parsing Table
- Input: Grammar G
- Output: Parsing table M

1. Get the set of productions of the grammar.
2. Initialize the parsing table P as a two dimensional structure with m x n enties (m is the number of non terminals in the grammar and n is the number of terminals + 1 for $).
3. For each production A→ x of the grammar do.
    1. For each terminal t in First (X)
        1. Add the production A → X to P[A, t].
    2. If epsilon is in First (X), then
        1. For each terminal S (including $ in Follow (A).
        2. Add A → ϵ to P[A, $].

PREDICTIVE PARSER

- Construction of Predictive parsing Table

4. All the undefined entries are error entries.
5. Whenever an entry to be added to the parsing table is already filled, then report that "Grammar is Ambiguous".

- **Example**

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \epsilon$

$T \rightarrow F\ T'$

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

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

PREDICTIVE PARSER

- **Example**

  E → T E'

  E' → + T E' | ϵ

  T → F T'

  T' → * F T' | ϵ

  F → (E) | id

- First (E) = {(, id}
- First (E') = {+, ϵ}
- First (T) = {(, id}
- First (T') = {*, ϵ}
- First (F) = {(, id}

Follow (E) = {), $}

Follow (E') = {), $}

Follow (T) = {+,), $}

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

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

PREDICTIVE PARSER

Predictive Parsing Table

Non Terminal	Input Symbol					
	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)		

PREDICTIVE PARSER

- **Example**

  $S \rightarrow i\ E\ t\ S\ S' \mid a$

  $S' \rightarrow e\ S \mid \epsilon$

  $E \rightarrow b$

- First (S) = {i, a}
- First (S') = {e, ϵ}
- First (E) = {b}
- Follow (S) = {e, $}
- Follow (S') = {e, $}
- Follow (E) = {t}

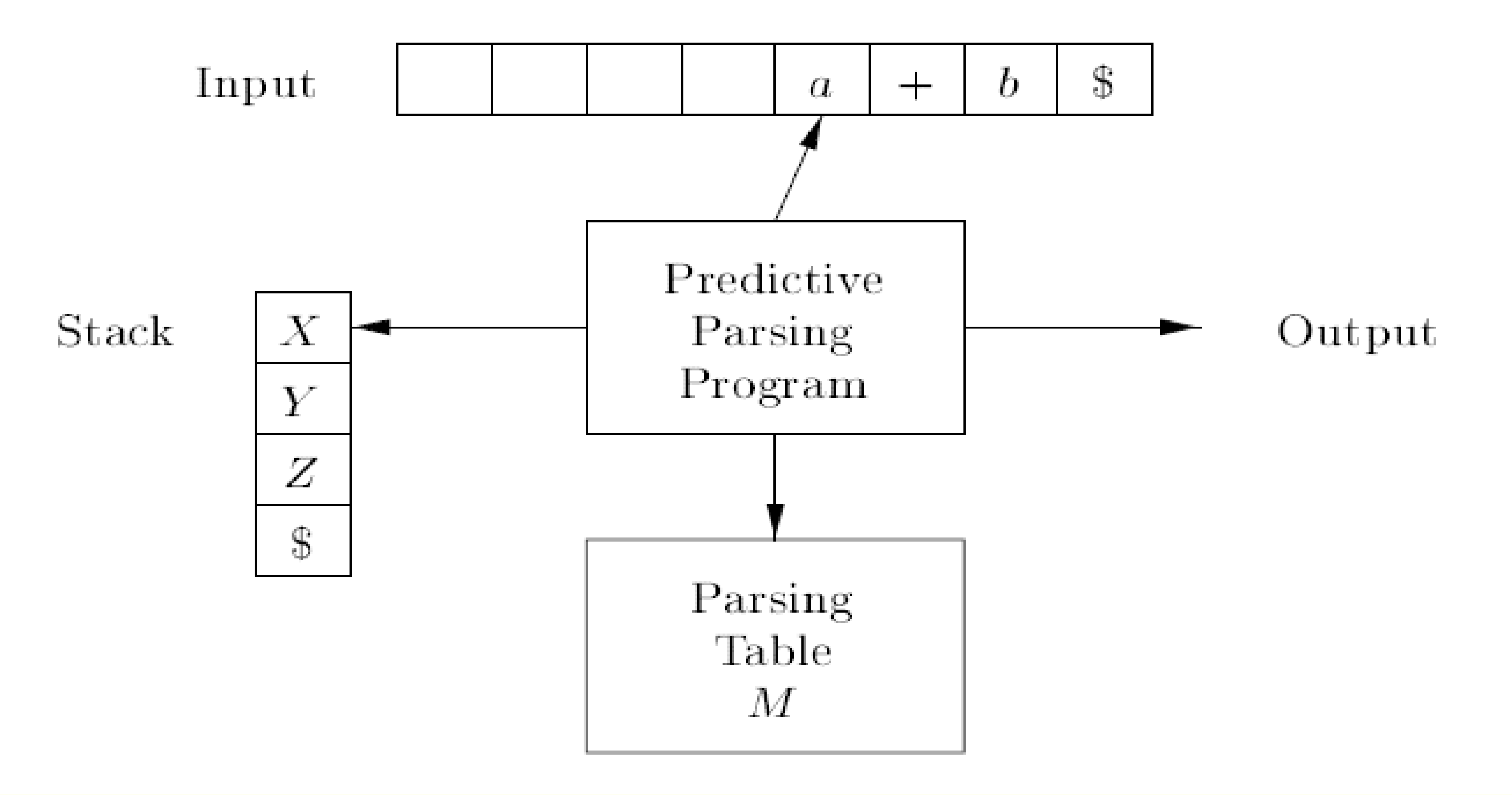
PREDICTIVE PARSER

Predictive Parsing Table

Non Terminal	Input Symbol					
	a	**b**	**e**	**i**	**t**	**$**
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Whenever an entry to be added to the parsing table is already filled, then report that "Grammar is Ambiguous".

PREDICTIVE PARSER – NON RECURSIVE – TABLE DRIVEN



PREDICTIVE PARSER – NON RECURSIVE – TABLE DRIVEN - ALG

1. α = top of stack & a – input symbol
2. If α = terminal then
    1. If α = a, then pop α and advance input pointer

       Else

       Invoke error();
3. If α = non-terminal α

- If M[α, a] → Y_1 Y_2 …,Y_n then pop α from stack pop Y_n … Y_2 Y_1 into stack such as Y is in top of stack.

  Else

  Invoke error routine.
- First of terminal is that terminal.
- First of non-terminal will be the first symbol its production.

PREDICTIVE PARSER – NON RECURSIVE – TABLE DRIVEN

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \epsilon$

$T \rightarrow F\ T'$

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

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

Non Terminal	**Input Symbol**					
	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

PREDICTIVE PARSER – NON RECURSIVE – TABLE DRIVEN

- **id + id * id $**

Stack	Input	Action
$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 $	id Matched Eliminate id from stack and input
$E’T’	+ id * id $	T’ → ϵ Eliminate T’
$E’	+ id * id $	E’ → +TE’
$E’T+	+ id * id $	+ Matched Eliminate + from stack and input

PREDICTIVE PARSER – NON RECURSIVE – TABLE DRIVEN

- **id + id * id $**

Stack	Input	Action
$E'T+	+ id * id $	+ Matched Eliminate + from stack and input
$E'T	id * id $	T → FT'
$E'T'F	id * id $	F → id
$E'T'id	id * id $	id Matched Eliminate id from stack and input
$E'T'	* id $	T' → *FT'
$E'T'F*	* id $	* Matched Eliminate * from stack and input
$E'T'F	id $	F → id

PREDICTIVE PARSER – NON RECURSIVE – TABLE DRIVEN

- **id + id * id $**

Stack	Input	Action
$E'T'F	id $	F → id
$E'T'id	id $	id Matched Eliminate id from stack and input
$E'T'	$	T' → ϵ Eliminate T'
$E'	$	E' → ϵ Eliminate E'
$	$	Final state Accepted
Correct Syntax		

PREDICTIVE PARSER – NON RECURSIVE – TABLE DRIVEN

- **id + * id $**

Stack	Input	Action
$E	id + * id $	E → TE’
$E’T	id + * id $	T → FT’
$E’T’F	id + * id $	F → id
$E’T’id	id + * id $	id Matched Eliminate id from stack and input
$E’T’	+ * id $	T’ → ϵ Eliminate T’
$E’	+ * id $	E’ → +TE’
$E’T+	+ * id $	+ Matched Eliminate + from stack and input

PREDICTIVE PARSER – NON RECURSIVE – TABLE DRIVEN

- **id + * id $**

Stack	Input	Action
$E'T+	+ * id $	+ Matched Eliminate + from stack and input
$E'T	* id $	Not Matched
Wrong Syntax		

ERROR RECOVERY PREDICTIVE PARSER

$E \rightarrow T E'$	First (E) = {(, id}	Follow (E) = {), \$}
$E' \rightarrow + T E' \mid \epsilon$	First (E') = {+, ϵ}	Follow (E') = {), \$}
$T \rightarrow F T'$	First (T) = {(, id}	Follow (T) = {+,), \$}
$T' \rightarrow * F T' \mid \epsilon$	First (T') = {*, ϵ}	Follow (T') = {+,), \$}
$F \rightarrow (E) \mid id$	First (F) = {(, id}	Follow (F) = {*, +,), \$}

Non Terminal	**Input Symbol**					
	id	**+**	*****	**(**	**)**	**\$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch

ERROR RECOVERY PREDICTIVE PARSER

- **) id * + id $**

Stack	Input	Action
$E	) id * + id $	Error) But synch so, skip and Eliminate)
$E	id * + id $	E → TE’
$E’T	id * + id $	T → FT’
$E’T’F	id * + id $	F → id
$E’T’id	id * + id $	id Matched Eliminate id from stack and input
$E’T’	* + id $	T’ → *FT’
$E’T’F*	* + id $	* Matched Eliminate * from stack and input

ERROR RECOVERY PREDICTIVE PARSER

- **) id * + id $**

Stack	Input	Action
$E'T'F*	* + id $	* Matched Eliminate * from stack and input
$E'T'F	+ id $	Error + But synch so, skip and Eliminate F
$E'T'	+ id $	T' → ϵ Eliminate T'
$E'	+ id $	E' → +TE'
$E'T+	+ id $	+ Matched Eliminate + from stack and input
$E'T	id $	T → FT'

ERROR RECOVERY PREDICTIVE PARSER

- **) id * + id $**

Stack	**Input**	**Action**
$E'T	id $	T → FT'
$E'T'F	id $	F → id
$E'T'id	id $	id Matched Eliminate id from stack and input
$E'T'	$	T' → ϵ Eliminate T'
$E'	$	E' → ϵ Eliminate E'
$	$	Accepted
But Error String		

BUTTOM-UP PARSING

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
- Bottom up parsing is also known as shift-reduce parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse.

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow id$

$F \rightarrow T$

$F \rightarrow id$

BUTTOM-UP PARSING

$E \rightarrow T$

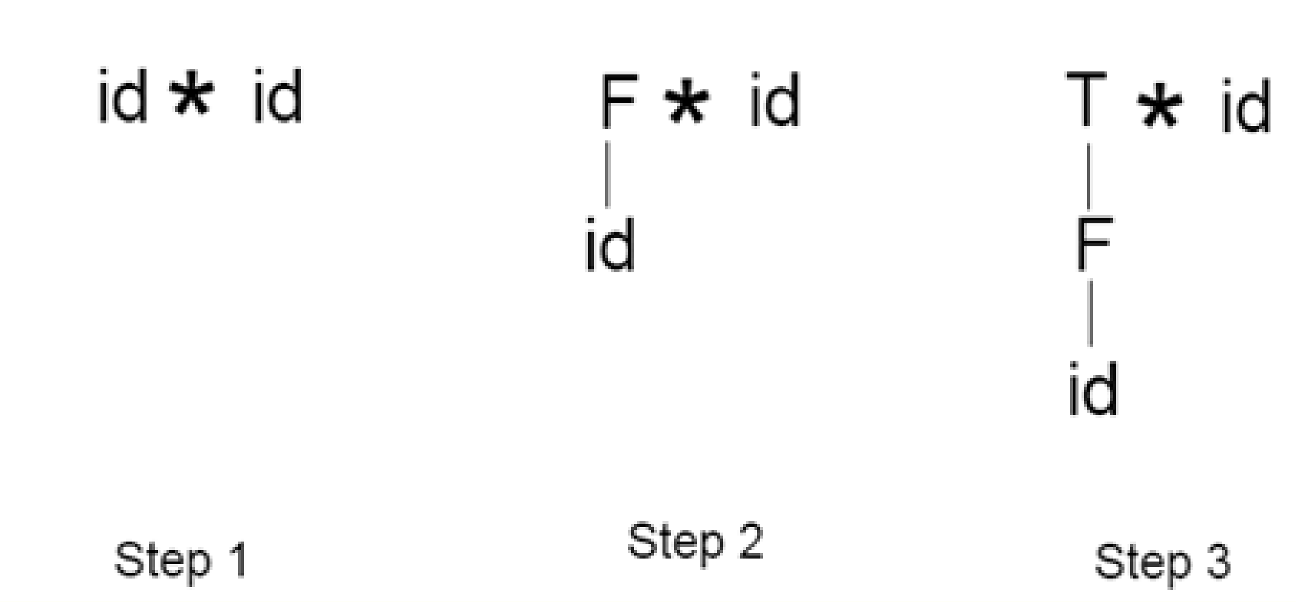
$T \rightarrow T * F$

$T \rightarrow id$

$F \rightarrow T$

$F \rightarrow id$

Input string id * id



BUTTOM-UP PARSING

$E \rightarrow T$

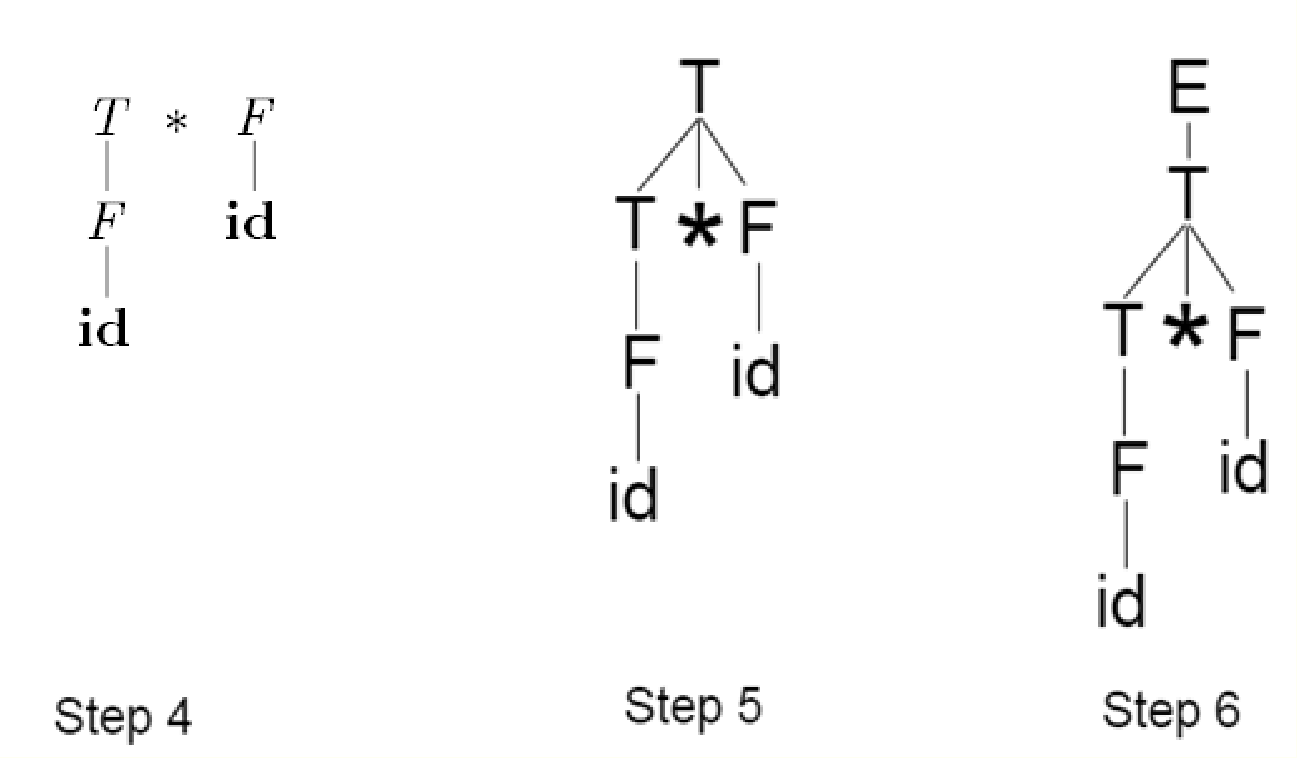
$T \rightarrow T * F$

$T \rightarrow id$

$F \rightarrow T$

$F \rightarrow id$

Input string id * id



BUTTOM-UP PARSING

- Bottom up parsing is classified in to various parsing. These are as follows:
- Shift-Reduce Parsing
- Operator Precedence Parsing
- Table Driven LR Parsing
  - LR(1)
  - SLR(1)
  - CLR (1)
  - LALR(1)

BUTTOM-UP PARSING – HANDLE PRUNING

- Bottom-up parsing during a left-to-right scan of the input constructs a right-most derivation in reverse.
- Handles puring a parse of $id_1 * id_2$

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 * id_2$	id_1	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$F * id_2$	Id_2	$F \rightarrow id$
$T * F$	$T * F$	$T \rightarrow T * F$
T	E	$E \rightarrow T$

SHIFT-REDUCE PARSING

- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- **4 possible actions**
  1. Shift
  2. Reduce
  3. Accept
  4. Error
- **Shift**
  - Shift the next input symbol onto the top of the stack.

SHIFT-REDUCE PARSING

- **Reduce**
  - The right end of the string to be reduced must be at the top of the stack.
  - Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
- **Accept**
  - Announce successful completion of parsing
- **Error**
  - Discover a syntax error and call an error recovery routine.

SHIFT-REDUCE PARSING

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

- Input ➔ $id_1 * id_2 \$$

Stack	Input	Action
$\$$	$id_1 * id_2 \$$	Shift
$\$ id_1$	$* id_2 \$$	Reduce $F \rightarrow id$
$\$ F$	$* id_2 \$$	Reduce $T \rightarrow F$
$\$ T$	$* id_2 \$$	Shift
$\$ T *$	$id_2 \$$	Shift
$\$ T * id_2$	$\$$	Reduce $F \rightarrow id$
$\$ T * F$	$\$$	Reduce $T \rightarrow T * F$
$\$ T$	$\$$	Reduce $E \rightarrow T$
$\$ E$	$\$$	Accept

OPERATOR PRECEDENCE PARSING

- Operator precedence grammar is kinds of shift reduce parsing method.
- It is applied to a small class of operator grammars.
- A grammar is said to be operator precedence grammar if it has two properties:
  - No R.H.S. of any production has a∈.
  - No two non-terminals are adjacent.
- Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal

OPERATOR PRECEDENCE PARSING

- **Rule-01:**
  - If precedence of b is higher than precedence of a, then we define a < b
  - If precedence of b is same as precedence of a, then we define a = b
  - If precedence of b is lower than precedence of a, then we define a > b
- **Rule-02:**
  - An identifier is always given the higher precedence than any other symbol.
  - $ symbol is always given the lowest precedence.
- **Rule-03:**
  - If two operators have the same precedence, then we go by checking their associativity.

OPERATOR PRECEDENCE PARSING

Precedence Table

	+	*	(	)	id	$
+	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	X
)	$\gtrdot$	$\gtrdot$	X	$\gtrdot$	X	$\gtrdot$
id	$\gtrdot$	$\gtrdot$	X	$\gtrdot$	X	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	X	$\lessdot$	X

OPERATOR PRECEDENCE PARSING

- **Precedence Table Algorithm**
- **Step-01:**
  - Insert the following-
  - $ symbol at the beginning and ending of the input string.
  - Precedence operator between every two symbols of the string by referring the operator precedence table.
- **Step-02:**
  - Start scanning the string from LHS in the forward direction until > symbol is encountered.
  - Keep a pointer on that location.

OPERATOR PRECEDENCE PARSING

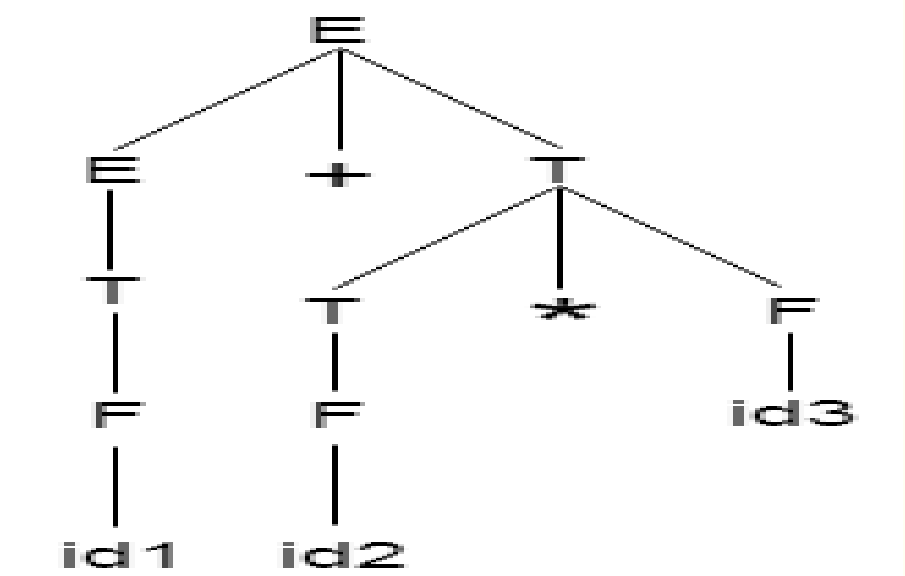
- **Precedence Table Algorithm**
- **Step-03:**
  - Start scanning the string from RHS in the backward direction until < symbol is encountered.
  - Keep a pointer on that location.
- **Step-04:**
  - Everything that lies in the middle of < and > forms the handle.
  - Replace the handle with the head of the respective production.
- **Step-05:**
  - Keep repeating the cycle from Step-02 to Step-04 until the start symbol is reached.

OPERATOR PRECEDENCE PARSING

- **Parsing Action**
  - Both end of the given input string, add the $ symbol.
  - Now scan the input string from left right until the $\gtrdot$ is encountered.
  - Scan towards left over all the equal precedence until the first left most $\lessdot$ is encountered.
  - Everything between left most $\lessdot$ and right most $\gtrdot$ is a handle.
  - $ on $ means parsing is successful.

OPERATOR PRECEDENCE PARSING

- E → E+T | T
- T → T*F | F
- F → id
- Input w = id + id * id



	E	T	F	id	+	*	$
E	X	X	X	X	$\doteq$	X	$\gtrdot$
T	X	X	X	X	$\gtrdot$	$\doteq$	$\gtrdot$
F	X	X	X	X	$\gtrdot$	$\gtrdot$	$\gtrdot$
id	X	X	X	X	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	X	$\doteq$	$\lessdot$	$\lessdot$	X	X	X
*	X	X	$\doteq$	$\lessdot$	X	X	X
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	X	X	X

OPERATOR PRECEDENCE PARSING

- $E \rightarrow E+T \mid T$
- $T \rightarrow T*F \mid F$
- $F \rightarrow id$
- Input w = id + id * id
- Now let us process the string with

the help of the above precedence table:

	E	T	F	id	+	*	$
E	X	X	X	X	$\doteq$	X	$\gtrdot$
T	X	X	X	X	$\gtrdot$	$\doteq$	$\gtrdot$
F	X	X	X	X	$\gtrdot$	$\gtrdot$	$\gtrdot$
id	X	X	X	X	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	X	$\doteq$	$\lessdot$	$\lessdot$	X	X	X
*	X	X	$\doteq$	$\lessdot$	X	X	X
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	X	X	X

$\$ \lessdot id1 \gtrdot + id2 * id3 \$$

$\$ \lessdot F \gtrdot + id2 * id3 \$$

$\$ \lessdot T \gtrdot + id2 * id3 \$$

$\$ \lessdot E \doteq + \lessdot id2 \gtrdot * id3 \$$

$\$ \lessdot E \doteq + \lessdot F \gtrdot * id3 \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \lessdot id3 \gtrdot \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \doteq F \gtrdot \$$

$\$ \lessdot E \doteq + \doteq T \gtrdot \$$

$\$ \lessdot E \doteq + \doteq T \gtrdot \$$

$\$ \lessdot E \gtrdot \$$

Accept.

OPERATOR PRECEDENCE PARSING

- Consider the following grammar-
- $E \rightarrow EAE \mid id$
- $A \rightarrow + \mid x$
- Construct the operator precedence parser and parse the string id + id x id.
- **Step-01:**
  - We convert the given grammar into operator precedence grammar.
  - The equivalent operator precedence grammar is-
  - $E \rightarrow E + E \mid E \times E \mid id$
- **Step-02:**
  - The terminal symbols in the grammar are { id, + , x , $ }
  - We construct the operator precedence table as-

OPERATOR PRECEDENCE PARSING

- Consider the following grammar-
- $E \rightarrow EAE \mid id$
- $A \rightarrow + \mid x$
- Construct the operator precedence parser and parse the string id + id x id.
- Operator Precedence Table

	id	+	x	$
id		>	>	>
+	<	>	<	>
x	<	>	>	>
$	<	<	<	

OPERATOR PRECEDENCE PARSING

- Parsing Given String-
- Given string to be parsed is id + id x id.
- We follow the following steps to parse the given string-
- **Step-01:**
  - We insert $ symbol at both ends of the string as-
  - $ id + id x id $
  - We insert precedence operators between the string symbols as-
  - $ < id > + < id > x < id > $

OPERATOR PRECEDENCE PARSING

- Parsing Given String-
- Given string to be parsed is id + id x id.
- Step-02:
- We scan and parse the string as-
  - $ < id > + < id > x < id > $
  - $ E + < id > x < id > $
  - $ E + E x < id > $
  - $ E + E x E $
  - $ + x $
  - $ < + < x > $
  - $ < + > $
  - $ $

E → EAE | id
A → + | x

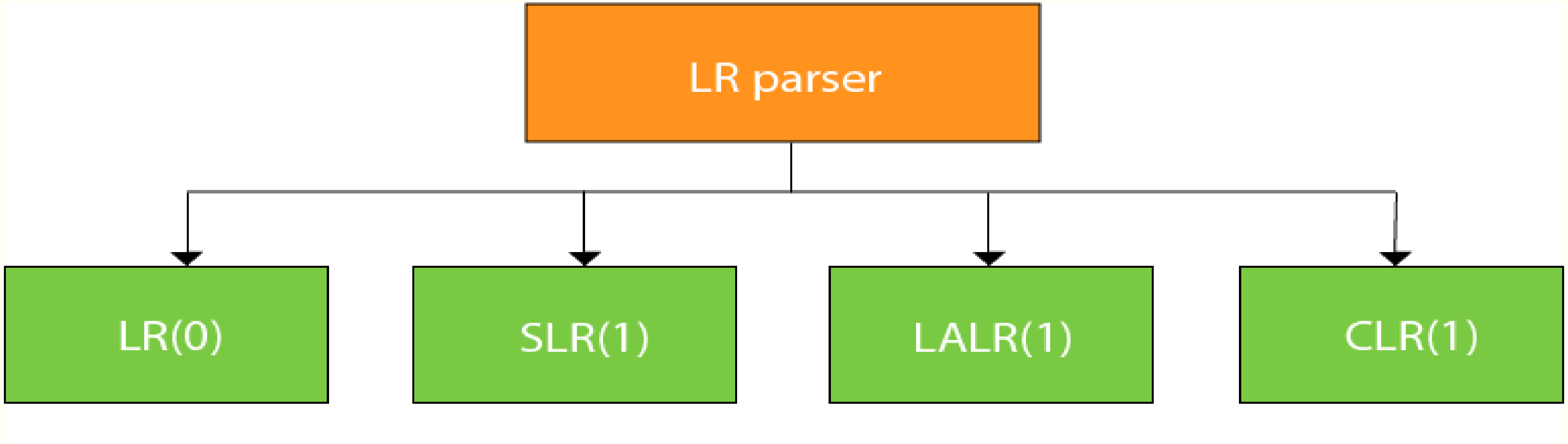
	id	+	x	$
id		>	>	>
+	<	>	<	>
x	<	>	>	>
$	<	<	<	

LR PARSING

- LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.
- In the LR parsing, "L" stands for left-to-right scanning of the input.
- "R" stands for constructing a right most derivation in reverse.
- "K" is the number of input symbols of the look ahead used to make number of parsing decision.
- LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing.

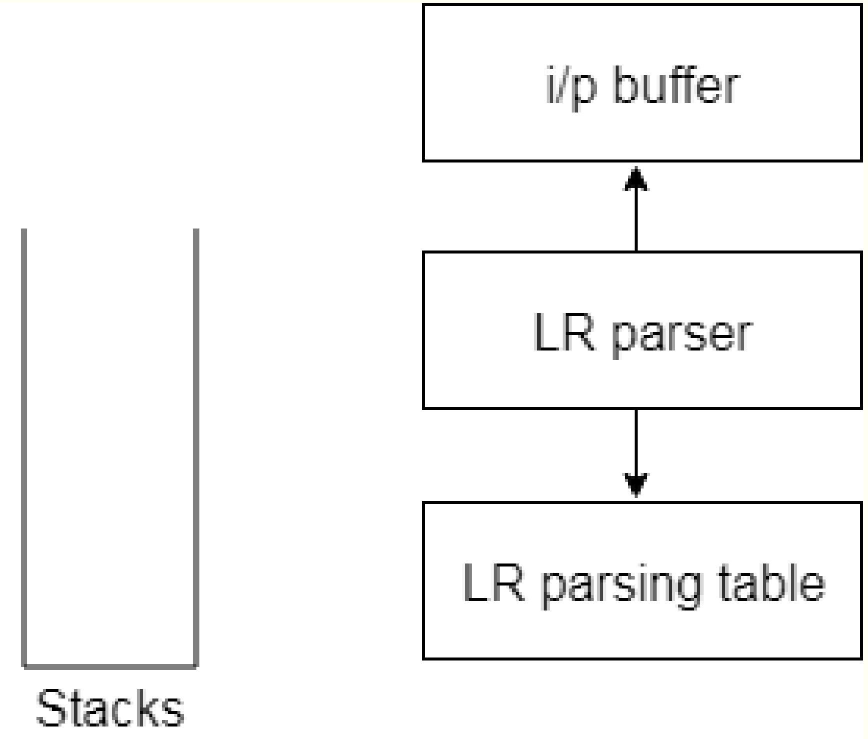
LR PARSING

Types of LR Parsing



LR PARSING

- **LR algorithm:**
- The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different.
- Input buffer is used to indicate end of input and it contains the string to be parsed followed by a $ Symbol.
- A stack is used to contain a sequence of grammar symbols with a $ at the bottom of the stack.
- Parsing table is a two dimensional array.
- It contains two parts: Action part and Go To part.



Block diagram of LR parser

LR PARSING

- **LR (1) Parsing**
- Various steps involved in the LR (1) Parsing:
- For the given input string write a context free grammar.
- Check the ambiguity of the grammar.
- Add Augment production in the given grammar.
- Create Canonical collection of LR (0) items.
- Draw a data flow diagram (DFA).
- Construct a LR (1) parsing table.
- **Example:**
- $S \rightarrow AA$
- $A \rightarrow aA \mid b$

The Augment grammar G` is represented by

$S` \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA \mid b$

LR PARSING

- Canonical Collection of LR(0) items
- An LR (0) item is a production G with dot at some position on the right side of the production.
- LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing.
- If I is a set of items for a grammar *G*, then *CLOSURE(I)* is the set of items constructed from I by the two rules:
  1. Initially, add every item in *I* to *CLOSURE(I)*.
  2. If $A \rightarrow \alpha \,.\, B\beta$ is in *CLOSURE(I)* and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow .\gamma$ to *CLOSURE(I)*, if it is not already there. Apply this rule until no more new items can be added to *CLOSURE(I)*.

LR PARSING

Canonical Collection of LR(0) items

1. Push all the items in I into the stack.
2. Add all the items in I to the closure C.
3. While stack not empty do
   a. Pop the top item in stack as T.
   b. If the item is of form $A \rightarrow \alpha \,.\, B\beta$ then
      1. For each production $B \rightarrow \gamma$ in the grammar do
         1. If $B \rightarrow .\gamma$ is not in C then
            a. Add $B \rightarrow .\gamma$ to C
            b. Push $B \rightarrow .\gamma$ to the stack.

E’ → . E , E → . E + T, E → . T , T → . T * F, T → . F, F → . (E), F → . id

LR PARSING

Find GOTO

1. Initialize an item J.
2. For each production of the form A → α .X β in I do

   a) If there is a production of the form A → α X. β then

3. Add closure (A → α X. β) to J.

   GOTO(I, X) = set of all items in J.

   GOTO(I0, E) = E' → E . , E → E. + T ➔ I1

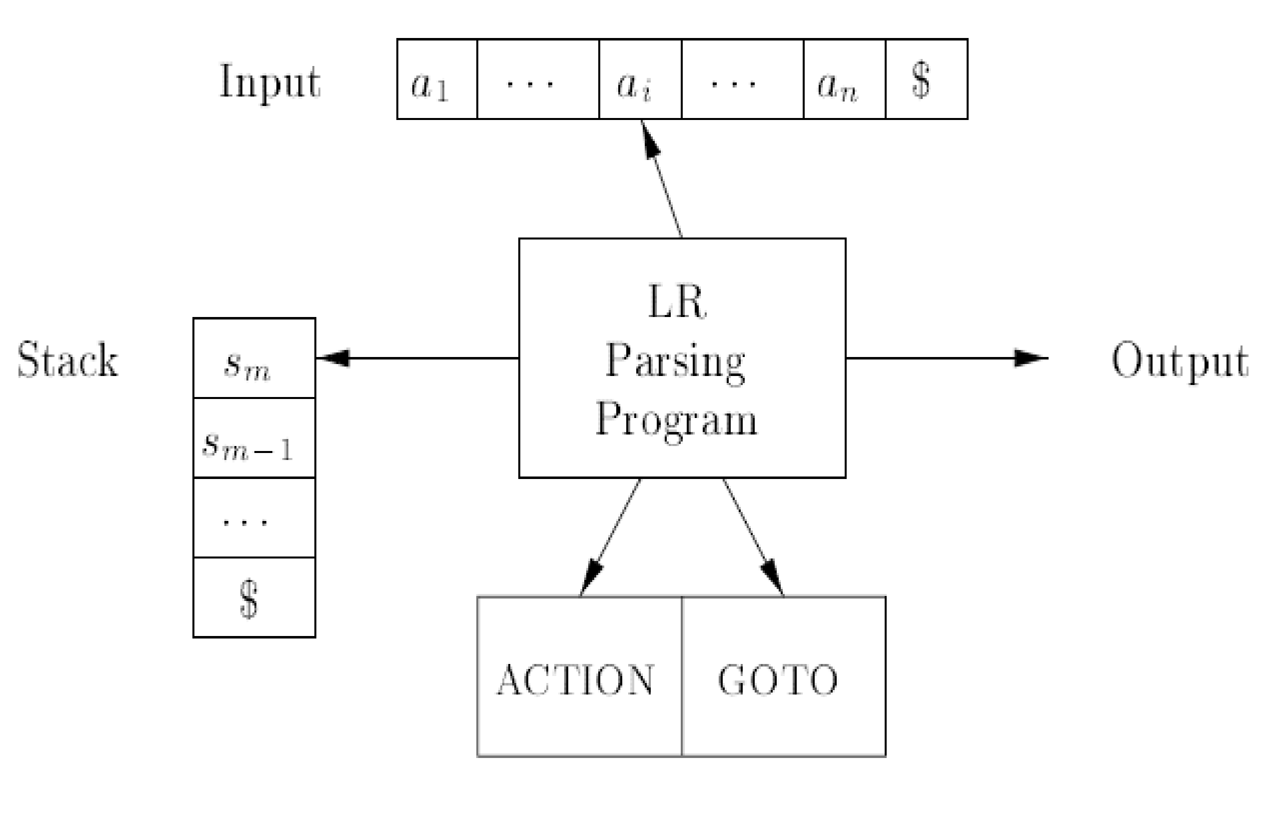
   GOTO(I0, T) = E → T. , T → T. * F ➔ I2

   GOTO(I0, F) = T → F. ➔ I3

   GOTO(I0,() = F→ (.E), E → .E + T ➔ I4

LR PARSING

- It consists of an input, an output, a stack, a driver program, and a parsing table has two parts (ACTION and GOTO).
- The driver program is the same for all LR parser.
- The parsing table changes from one parser to another.
- The parsing program reads characters from an input buffer one at a time.
- Where a shift-reduce parser would shift a symbol, an LR parser shifts a state.



LR PARSING

- **Example**

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

  $F \rightarrow (E) \mid id$

- Write with numbers

  1. $E \rightarrow E + T$
  2. $E \rightarrow T$
  3. $T \rightarrow T * F$
  4. $T \rightarrow F$
  5. $F \rightarrow (E)$
  6. $F \rightarrow id$

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

  Follow (E) = {), $}
  Follow (E’) = {), $}
  Follow (T) = {+,), $}
  Follow (T’) = {+,), $}
  Follow (F) = {*, +,), $}

- Add Augment production in the given grammar. **E’ → E**

LR PARSING

- New production

  E' → E

  E → E + T

  E → T

  T →T * F

  T → F

  F → (E)

  F → id
- Create Canonical collection of LR (0) items.
- An LR (0) item is a production G with dot at some position on the right side of the production.

LR PARSING

- Create Canonical collection of LR (0) items.
- An LR (0) item is a production G with dot at some position on the right side of the production. **Closure – LR (0) Items**

E' → . E
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I0

GOTO (I0, E)
E' → E .
E → E . + T

I1

GOTO (I0, T)
E → T .
T → T . * F

I2

GOTO (I0, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I4

GOTO (I0, F)
T → F . **→ I3**

GOTO (I0, id)
F → id . **→ I5**

LR PARSING

GOTO (I1, +)

E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

I6

GOTO (I2, *)

T → T * . F
F → . (E)
F → . id

I7

GOTO (I4, E)

F → (E .)
E → E . + T

I8

GOTO (I4, ()

F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I4

GOTO (I4, T)

E → T .
T → T . * F

I2 (Already)

GOTO (I4, F)

T → F . ➔ **I3**

LR PARSING

GOTO (I4, id)
F → id . ➔ **I5**

GOTO (I6, T)
E → E + T .
T → T . * F
I9

GOTO (I6, F)
T → F . ➔ **I3**

GOTO (I6, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id
I4

GOTO (I6, id)
F → id . ➔ **I5**

GOTO (I7, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id
I4

GOTO (I7, F)
T → T * F . ➔ **I10**

GOTO (I7, id)
F → id . ➔ **I5**

GOTO (I8,))
F → (E) . ➔ **I11**

LR PARSING

GOTO (I8, +)

E → E + . T
T → T . * F
T → . F
F → . (E)
F → . id

I6

GOTO (I9, *)

T → T * . F
T → . F
F → . (E)
F → . id

I7

LR PARSING

Two Classes

1. **Kernel items**

The initial item, S' → . S, and all items whose dots are not at the left end.

2. **Nonkernel items**

All items with their dots at the left end, except for S' → . S

F → (. E) ➔ kernel

E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

Non kernel

LR PARSING TABLE ALGORITHM

Two Parts

1. ACTION
2. GOTO

1. The ACTION function takes as arguments a state i and a terminal a(or $, the input endmarker).

→ The value of ACTION(i, a) can have one of four forms.

a) Shift j where j is a state.

→ The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a.

b) Reduce A → β.

→ The action of the parser effectively reduces β on the top of the stack to head A.

LR PARSING TABLE

c) Accept.

→ The parser accepts the input and finishes parsing.

d) Error.

→ The parser discovers an error in its input and takes some corrective action.

2. The GOTO function, defined on sets of items, to states.

If GOTO (I_i, A) = I_j, then GOTO also maps a state i and a nonterminal A to state j.

LR PARSING TABLE

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2	R2	R2	S7	R2	R2	R2			
3	R4	R4	R4	R4	R4	R4			
4	S5			S4			8	2	3
5	R6	R6	R6	R6	R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9	R1	R1	S7	R1	R1	R1			
10	R3	R3	R3	R3	R3	R3			
11	R5	R5	R5	R5	R5	R5			

1) E → E + T, 2) E → T, 3) T → T * F, 4) T → F, 5) F → (E), 6) F → id

LR PARSING TABLE

- Input id * id + id $

Stack	Symbols	Input	Action
0		id * id + id $	0 → id → Shift S5 Push 5 and shift input id
0 5 (pop)	id	* id + id $	5 → * → R6 (F → id) Pop 5 and Reduce id by F
0	F	* id + id $	0 → F → 3 Push 3
0 3 (pop)	F	* id + id $	3 → * → R4 (T→ F) Pop 3 and Reduce F by T
0	T	* id + id $	0 → T → 2 Push 2
0 2	T	* id + id $	2 → * → S7 Push 7 and shift input *
0 2 7	T *	id + id $	7 → id → S5 Push 5 and shift input id

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2	R2	R2	S7	R2	R2	R2			
3	R4	R4	R4	R4	R4	R4			
4	S5			S4			8	2	3
5	R6	R6	R6	R6	R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9	R1	R1	S7	R1	R1	R1			
10	R3	R3	R3	R3	R3	R3			
11	R5	R5	R5	R5	R5	R5			

1) E → E + T, 2) E → T, 3) T → T * F, 4) T → F, 5) F → (E), 6) F → id

LR PARSING TABLE

- Input id * id + id $

Stack	Symbols	Input	Action
0 2 7	T *	id + id $	7 → id → S5 Push 5 and shift input id
0 2 7 5 (pop)	T * id	+ id $	5 → + → R6 (F → id) Pop 5 and Reduce id by F
0 2 7	T * F	+ id $	7 → F → 10 Push 10
0 2 7 10 (pop)	T * F	+ id $	10 → + → R3 (T→ T * F) Pop 10, 7, 2 and Reduce T * F by T
0	T	+ id $	0 → T → 2 Push 2
0 2 (pop)	T	+ id $	2 → + → R2 (E → T) Pop 2 and Reduce T by E
0	E	+ id $	0 → E → 1

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2	R2	R2	S7	R2	R2	R2			
3	R4	R4	R4	R4	R4	R4			
4	S5			S4			8	2	3
5	R6	R6	R6	R6	R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9	R1	R1	S7	R1	R1	R1			
10	R3	R3	R3	R3	R3	R3			
11	R5	R5	R5	R5	R5	R5			

1) E → E + T, 2) E → T, 3) T → T * F, 4) T → F, 5) F → (E), 6) F → id

LR PARSING TABLE

- Input id * id + id $

Stack	Symbols	Input	Action
0	E	+ id $	0 → E → 1 Push 1
0 1	E	+ id $	1 → + → S6 Push 6 and shift input +
0 1 6	E +	id $	6 → id → S5 Push 5 and shift input id
0 1 6 5	E + id	$	5 → $ → R6 (F → id) Pop 5 and Reduce id by F
0 1 6	E + F	$	6 → F → 3 Push 3
0 1 6 3	E + F	$	3 → $ → R4 (T → F) Pop 3 and Reduce F by T
0 1 6	E + T	$	6 → T → 9 Push 9

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2	R2	R2	S7	R2	R2	R2			
3	R4	R4	R4	R4	R4	R4			
4	S5			S4			8	2	3
5	R6	R6	R6	R6	R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9	R1	R1	S7	R1	R1	R1			
10	R3	R3	R3	R3	R3	R3			
11	R5	R5	R5	R5	R5	R5			

1) E → E + T, 2) E → T, 3) T → T * F, 4) T → F, 5) F → (E), 6) F → id

LR PARSING TABLE

- Input id * id + id $

Stack	Symbols	Input	Action
0 1 6	E + T	$	6 → T → 9 Push 9
0 1 6 9	E + T	$	9 → $ → R1 (E → E + T) Pop 9, 6, 1 and Reduce E + T by E
0	E	$	0 → E → 1 Push 1
0 1	E	$	1 → E → Accept

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2	R2	R2	S7	R2	R2	R2			
3	R4	R4	R4	R4	R4	R4			
4	S5			S4			8	2	3
5	R6	R6	R6	R6	R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9	R1	R1	S7	R1	R1	R1			
10	R3	R3	R3	R3	R3	R3			
11	R5	R5	R5	R5	R5	R5			

SLR PARSING TABLE ALGORITHM

- Input: An augmented grammar G’
- Output: The SLR parsing table functions ACTION and GOTO for G’.
- Method

1. Construct C = $\{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G’.
2. State i is constructed from I_i.
   - The parsing actions for state i are determined as follows.

     a) If ($A \rightarrow \alpha.\ A\beta$) is in I_i and GOTO (I_i, a) = I_i, then set ACTION (i, a) to “shift j”.

     Here a must be a terminal.

     b) If ($A \rightarrow \alpha.$) is in I_i, then set ACTION (i,a) to “reduce $A \rightarrow \alpha$” for all a in Follow(A);

     Here A may not be S’.

SLR PARSING TABLE ALGORITHM

c) If (S' ➔ S.) is in I_i, then set ACTION (i, $) to "accept".

- If any conflicting actions result from above rules, we say the grammar is not SLR(1).
- The algorithm fails to produce a parser in this case.

3. The goto transitions for state i are constructed for all nonterminals A using the rule:

   If GOTO (Ii, A) = Ij then

   GOTO (I, A) = j.

3. All entries not defined by rules (2) and (3) are made "error".
4. The initial state of the parser is the one constructed from the set of items containing (S' ➔ .S).

SLR PARSING

- **Example**

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

  $F \rightarrow (E) \mid id$

- Write with numbers

  1. $E \rightarrow E + T$
  2. $E \rightarrow T$
  3. $T \rightarrow T * F$
  4. $T \rightarrow F$
  5. $F \rightarrow (E)$
  6. $F \rightarrow id$

First (E) = {(, id}
First (E') = {+, ϵ}
First (T) = {(, id}
First (T') = {*, ϵ}
First (F) = {(, id}

Follow (E) = {), $}
Follow (E') = {), $}
Follow (T) = {+,), $}
Follow (T') = {+,), $}
Follow (F) = {*, +,), $}

- Add Augment production in the given grammar. **$E' \rightarrow E$**

SLR PARSING

- New production

  $E' \rightarrow E$

  $E \rightarrow E + T$

  $E \rightarrow T$

  $T \rightarrow T * F$

  $T \rightarrow F$

  $F \rightarrow (E)$

  $F \rightarrow id$

- Create Canonical collection of LR (0) items.
- An LR (0) item is a production G with dot at some position on the right side of the production.

SLR PARSING

- Create Canonical collection of LR (0) items.
- An LR (0) item is a production G with dot at some position on the right side of the production. **Closure – LR (0) Items**

E’ → . E
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I0

GOTO (I0, E)
E’ → E .
E → E . + T

I1

GOTO (I0, T)
E → T .
T → T . * F

I2

GOTO (I0, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I4

GOTO (I0, F)
T → F . **→ I3**

GOTO (I0, id)
F → id . **→ I5**

SLR PARSING

GOTO (I1, +)

E → E + . T
T → T . * F
T → . F
F → . (E)
F → . id

I6

GOTO (I2, *)

T → T * . F
T → . F
F → . (E)
F → . id

I7

GOTO (I4, E)

F → (E .)
E → E . + T

I8

GOTO (I4, ()

F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I4

GOTO (I4, T)

E → T .
T → T . * F

I2 (Already)

GOTO (I4, F)

T → F . → **I3**

SLR PARSING

GOTO (I4, id)

F → id . ➔ **I5**

GOTO (I6, T)

E → E + T .

T → T . * F

I9

GOTO (I6, F)

T → F . ➔ **I3**

GOTO (I6, ()

F → (. E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → . id

I4

GOTO (I6, id)

F → id . ➔ **I5**

GOTO (I7, ()

F → (. E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → . id

I4

GOTO (I7, F)

T → T * F . ➔ **I10**

GOTO (I7, id)

F → id . ➔ **I5**

GOTO (I8,))

F → (E) . ➔ **I11**

SLR PARSING

GOTO (I8, +)

E → E + . T
T → T . * F
T → . F
F → . (E)
F → . id

I6

GOTO (I9, *)

T → T * . F
T → . F
F → . (E)
F → . id

I7

SLR PARSING TABLE

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1) E → E + T, 2) E → T, 3) T → T * F, 4) T → F, 5) F → (E), 6) F → id

SLR PARSING TABLE

- Input id * id + id $

Stack	Symbols	Input	Action
0		id * id + id $	0 → id → Shift S5 Push 5 and shift input id
0 5 (pop)	id	* id + id $	5 → * → R6 (F → id) Pop 5 and Reduce id by F
0	F	* id + id $	0 → F → 3 Push 3
0 3 (pop)	F	* id + id $	3 → * → R4 (T→ F) Pop 3 and Reduce F by T
0	T	* id + id $	0 → T → 2 Push 2
0 2	T	* id + id $	2 → * → S7 Push 7 and shift input *
0 2 7	T *	id + id $	7 → id → S5 Push 5 and shift input id

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1) E → E + T, 2) E → T, 3) T → T * F, 4) T → F, 5) F → (E),
6) F → id

SLR PARSING TABLE

- Input id * id + id $

Stack	Symbols	Input	Action
0 2 7	T *	id + id $	7 → id → S5 Push 5 and shift input id
0 2 7 5 (pop)	T * id	+ id $	5 → + → R6 (F → id) Pop 5 and Reduce id by F
0 2 7	T * F	+ id $	7 → F → 10 Push 10
0 2 7 10 (pop)	T * F	+ id $	10 → + → R3 (T→ T * F) Pop 10, 7, 2 and Reduce T * F by T
0	T	+ id $	0 → T → 2 Push 2
0 2 (pop)	T	+ id $	2 → + → R2 (E → T) Pop 2 and Reduce T by E
0	E	+ id $	0 → E → 1

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1) E → E + T, 2) E → T, 3) T → T * F, 4) T → F, 5) F → (E),
6) F → id

SLR PARSING TABLE

- Input id * id + id $

Stack	Symbols	Input	Action
0	E	+ id $	0 → E → 1 Push 1
0 1	E	+ id $	1 → + → S6 Push 6 and shift input +
0 1 6	E +	id $	6 → id → S5 Push 5 and shift input id
0 1 6 5	E + id	$	5 → $ → R6 (F → id) Pop 5 and Reduce id by F
0 1 6	E + F	$	6 → F → 3 Push 3
0 1 6 3	E + F	$	3 → $ → R4 (T → F) Pop 3 and Reduce F by T
0 1 6	E + T	$	6 → T → 9 Push 9

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1) E → E + T, 2) E → T, 3) T → T * F, 4) T → F, 5) F → (E), 6) F → id

SLR PARSING TABLE

- Input id * id + id $

Stack	Symbols	Input	Action
0 1 6	E + T	$	6 → T → 9 Push 9
0 1 6 9	E + T	$	9 → $ → R1 (E → E + T) Pop 9, 6, 1 and Reduce E + T by E
0	E	$	0 → E → 1 Push 1
0 1	E	$	1 → E → Accept

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1) E → E + T, 2) E → T, 3) T → T * F, 4) T → F, 5) F → (E), 6) F → id

SLR PARSING TABLE

- Input id * id $

Stack	Symbols	Input	Action
0		id * id $	0 → id → Shift S5 Push 5 and shift input id
0 5 (pop)	id	* id $	5 → * → R6 (F → id) Pop 5 and Reduce id by F
0	F	* id $	0 → F → 3 Push 3
0 3 (pop)	F	* id $	3 → * → R4 (T→ F) Pop 3 and Reduce F by T
0	T	* id $	0 → T → 2 Push 2
0 2	T	* id $	2 → * → S7 Push 7 and shift input *
0 2 7	T *	id $	7 → id → S5 Push 5 and shift input id

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1) E → E + T, 2) E → T, 3) T → T * F, 4) T → F, 5) F → (E), 6) F → id

SLR PARSING TABLE

- Input id * id $

Stack	Symbols	Input	Action
0 2 7	T *	id $	7 → id → S5 Push 5 and shift input id
0 2 7 5 (pop)	T * id	$	5 → + → R6 (F → id) Pop 5 and Reduce id by F
0 2 7	T * F	$	7 → F → 10 Push 10
0 2 7 10 (pop)	T * F	$	10 → $ → R3 (T→ T * F) Pop 10, 7, 2 and Reduce T * F by T
0	T	$	0 → T → 2 Push 2
0 2 (pop)	T	$	2 → $ → R2 (E → T) Pop 2 and Reduce T by E
0	E	$	0 → E → 1

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1) E → E + T, 2) E → T, 3) T → T * F, 4) T → F, 5) F → (E), 6) F → id

SLR PARSING TABLE

- Input id * id $

Stack	Symbols	Input	Action
0	E	$	0 → E → 1 Push 1
0 1	E	$	1 → $ → Accept

State	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

SLR PARSING TABLE

- Example

  $S \rightarrow L = R \mid R$

  $L \rightarrow * R \mid id$

  $R \rightarrow L$

- Production with numbers

  1. $S \rightarrow L = R$
  2. $S \rightarrow R$
  3. $L \rightarrow * R$
  4. $L \rightarrow id$
  5. $R \rightarrow L$

SLR PARSING TABLE

- Add Augment production in the given grammar. **S' → S**

  S' → S

  S → L = R

  S → R

  L → * R

  L → id

  R → L

First (S) = {*, id}
First (L) = {*, id}
First (R) = {*, id}

Follow (S) = {$}
Follow (L) = {=, $}
Follow (R) = {=, $}

- Create Canonical collection of LR (0) items.
- An LR (0) item is a production G with dot at some position on the right side of the production.

SLR PARSING TABLE

- Canonical collection of LR (0) items

S' → .S
S → .L = R
S → .R
L → .* R
L → .id
R → .L

I0

GOTO (I0, S)
S' → S. **→ I1**

GOTO (I0, L)
S → L. = R
R → L. **→ I2**

GOTO (I0, R)
S → R. **→ I3**

GOTOT (I0, *)
L → *. R
R → .L
L → .* R
L → .id

I4

GOTO (I0, id)
L → id. **→ I5**

GOTOT (I2, =)
S → L =. R
R → .L
L → .* R
L → .id

I6

SLR PARSING TABLE

GOTO (I4, R)
L → * R. **→ I7**

GOTO (I4, L)
R → L. **→ I8**

GOTO (I6, R)
S → L = R. **→ I9**

GOTOT (I4, *)
L → *. R
R → .L
L → .* R **→ I4**
L → .id

GOTO (I4, id)
L → id. **→ I5**

GOTO (I6, L)
R → L. **→ I8**

GOTOT (I6, *)
L → *. R
R → .L
L → .* R **→ I4**
L → .id

GOTO (I6, id)
L → id. **→ I5**

SLR PARSING TABLE

State	Action				GOTO		
	*	=	id	$	S	L	R
0	S4		S5		1	2	3
1				Accept			
2	S4	R5 / S6	S5	R5			
3				R2			
4	S4		S5			8	7
5		R4		R4			
6	S4		S5			8	9
7		R3		R3			
8		R5		R5			
9				R1			

SLR PARSING TABLE

1) S → L = R 2) S → R
3) L → * R 4) L → id
5) R → L

- Input id = * id $

Stack	Symbols	Input	Action
0		id = * id $	0 → id → S5 Push 5 and shift input id
0 5 Pop	id	= * id $	5 → id → R4 (L → id Pop 5 and Reduce id by L
0	L	= * id $	0 → L → 2 Push 2
0 2 pop	L	= * id $	2 → L → R5 / S5 Reduce R → L
0	R	= * id $	3 → = → No Production Not accept

State	Action				GOTO		
	*	=	id	$	S	L	R
0	S4		S5		1	2	3
1				Accept			
2	S4	R5 / S6	S5	R5			
3				R2			
4	S4		S5			8	7
5		R4		R4			
6	S4		S5			8	9
7		R3		R3			
8		R5		R5			
9				R1			

PARSING PROBLEMS

- **Example 1**
  - S → A a A b | B b B a
  - A →
  - B →
  - LL(1) but not SLR (1).
- **Example 2**
  - S → S A | A
  - A → a
  - Is SLR(1) but not LL(1).

REFERENCE

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, “Compilers: Principles, Techniques and Tools”, Second Edition, Pearson Education Limited, 2014.
2. Randy Allen, Ken Kennedy, “Optimizing Compilers for Modern Architectures: A Dependence-based Approach”, Morgan Kaufmann Publishers, 2002.
3. Steven S. Muchnick, “Advanced Compiler Design and Implementation”, Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint, 2003.
4. Keith D Cooper and Linda Torczon, “Engineering a Compiler”, Morgan Kaufmann Publishers, Elsevier Science, 2004.
5. V. Raghavan, “Principles of Compiler Design”, Tata McGraw Hill Education Publishers, 2010.
6. Allen I. Holub, “Compiler Design in C”, Prentice-Hall Software Series, 1993.


Thank you