Syntax Analysis

Mrs. Sunita M Dol

(sunitaaher@gmail.com)

Assistant Professor, Dept of Computer Science and Engineering

Walchand Institute of Technology, Solapur
(www.witsolapur.org)

Syntax Analysis

- Role of Parser
- Writing grammars for context free environments
- Top-down parsing
- Recursive descent and predictive parsers (LL)
- Bottom-Up parsing
- Operator precedence parsing
- LR parsers
- SLR parsers

The Role of the Parser

- Parser for any grammar is program that takes as input string w (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for w , if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar.
- The goal of the parser is to determine the syntactic validity of a source string is valid, a tree is built for use by the subsequent phases of the computer.

The Role of the Parser

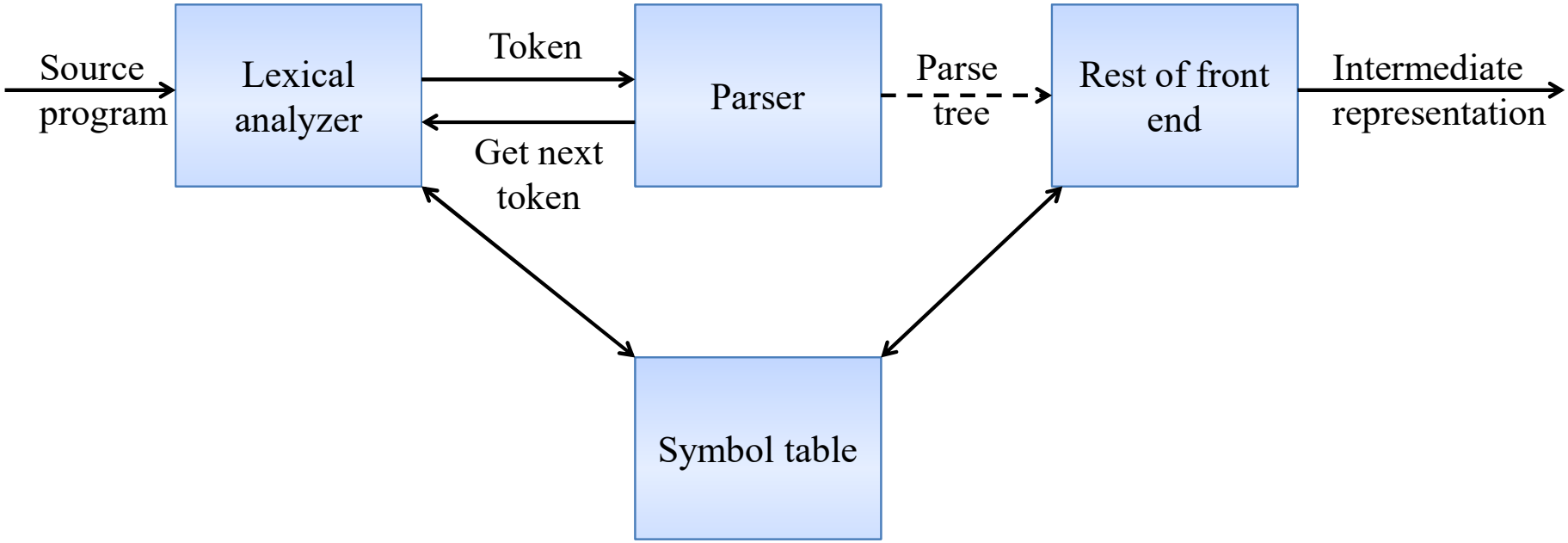


Fig: Position of parser in compiler model

The Role of the Parser

- **Types of parsers**
  - ✓ Top-down parser - which build parse trees from top(root) to bottom(leaves)
  - ✓ Bottom-up parser - which build parse trees from leaves and work up the root.

The Role of the Parser

- **Syntax error handling**
  - ✓ **Lexical**: e.g. misspelling an identifier
  - ✓ **Syntactic**: e.g. arithmetic expression with unbalanced parentheses
  - ✓ **Semantic**: e.g. operator applied to incompatible operand
  - ✓ **Logical**: e.g. infinitely recursive call

  Error handler in parsers

  - ✓ Should report the presence of errors clearly and accurately
  - ✓ Should recover from each error quickly
  - ✓ Should not significantly slow down the processing of correct program

The Role of the Parser

- **Error recovery strategies**
  - ✓ **Panic mode**: Parser discards input symbols one at a time until one of a designated set of synchronizing(e.g. ';') token is found.
  - ✓ **Phrase level**: Parser performs local correction on the remaining input.
  - ✓ **Error productions**: We augment the error productions to construct a parser. Error diagnostics can be generated to indicate the erroneous construct
  - ✓ **Global correction**: Minimal sequence of changes to obtain a globally least cost correction

Think and Write

Identify the types of error in the following examples

a. 12ab

b. (a+(b*c)

c. 2 + a[i]

d. for (;;)

```
{
        printf("Hello\n");
}
```

Think and Write

Identify the types of error in the following examples

a. 12ab – **Lexical error**

b. (a+(b*c) – **Syntactic error**

c. 2 + a[i] – **Semantic error**

d. for (;;)

```
{
        printf("Hello\n");
}
```

Logical error

Context Free Grammars

- Inherently recursive structures of a programming language
- In a context-free grammar, we have:
  - ✓ A finite set of terminals
  - ✓ A finite set of non-terminals (syntactic-variables)
  - ✓ A finite set of productions rules in the form $A \rightarrow \alpha$
  - ✓ A start symbol (one of the non-terminal symbol)

Grammar for Simple arithmetic expression

expr → expr op expr	expr → (expr)	expr → - expr
expr → id	op → +	op → -
op → *	op → /	op → ↑

Context Free Grammars

- **Notational Conventions**
  - ✓ **Terminals**
    - ➢ Lower case letters early in the alphabets
    - ➢ Operator symbols
    - ➢ Punctuation symbols such as parentheses, comma, etc.
    - ➢ Digits
    - ➢ Boldface strings **id** or **if**
  - ✓ **Nonterminals**
    - ➢ Uppercase letters early in the alphabet
    - ➢ The letter S, start symbol
    - ➢ Lower case italic names such as ***expr*** or ***stmt***

Context Free Grammars

- **Notational Conventions**
  - ✓ Upper case letter late in alphabet – grammar symbols (terminals or nonterminals)
  - ✓ Lower case letter late in alphabet – strings of terminals
  - ✓ Lower case Greek letters – strings of grammar symbols
  - ✓ If $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots, A \rightarrow \alpha_k$ then $A \rightarrow \alpha 1 | \alpha 2 | \ldots | \alpha_k$
  - ✓ Left side of production – start symbol

Grammar for Simple arithmetic expression

$E \rightarrow E\ A\ E \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

Context Free Grammars

- **Derivations:** a sequence of replacements of non-terminal symbols
  - ✓ In general a derivation step is $\alpha A \beta \Rightarrow \alpha\gamma\beta$ if there is a production rule $A \rightarrow \gamma$ in our grammar where α and β are arbitrary strings of terminal and non-terminal symbols
    - ➢ $\Rightarrow$: derives in one step
    - ➢ $\overset{*}{\Rightarrow}$: derives in zero or more steps
    - ➢ $\overset{+}{\Rightarrow}$: derives in one or more steps

Grammar for Simple arithmetic expression

E → E A E | (E) | -E | id

A → + | - | * | / | ↑

Derivation of id+id

E → E A E

→ id A E

→ id + E

→ id + id

Context Free Grammars

- **Derivations:**
  - ✓ **Leftmost derivation**
  - ✓ **Rightmost derivation**

Grammar for Simple arithmetic expression

$E \rightarrow E\ A\ E \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

Leftmost Derivation of id+id

$E \rightarrow E\ A\ E$
$\rightarrow id\ A\ E$
$\rightarrow id + E$
$\rightarrow id + id$

Rightmost Derivation of id+id

$E \rightarrow E\ A\ E$
$\rightarrow E\ A\ id$
$\rightarrow E + id$
$\rightarrow id + id$

Context Free Grammars

- **Parse Tree :** A parse tree can be seen as a graphical representation of a derivation.
  - ✓ Inner nodes of a parse tree are non-terminal symbols.
  - ✓ The leaves of a parse tree are terminal symbols.

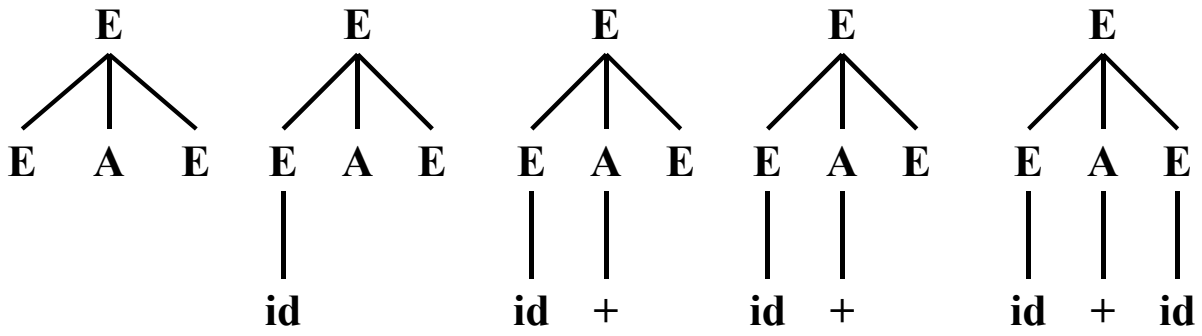
Grammar for Simple arithmetic expression

E → E A E | (E) | -E | id

A → + | - | * | / | ↑

Leftmost Derivation of id+id

E → E A E
→ id A E
→ id + E
→ id + id



Context Free Grammars

- **Parse Tree :** A parse tree can be seen as a graphical representation of a derivation.

Grammar for Simple arithmetic expression

E → E A E | (E) | -E | id

A → + | - | * | / | ↑

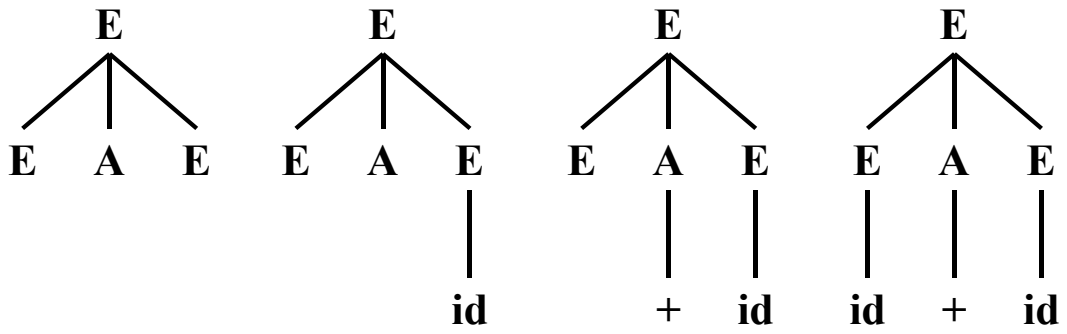
Rightmost Derivation of id+id

E → E A E

→ E A id

→ E + id

→ id + id



Context Free Grammars

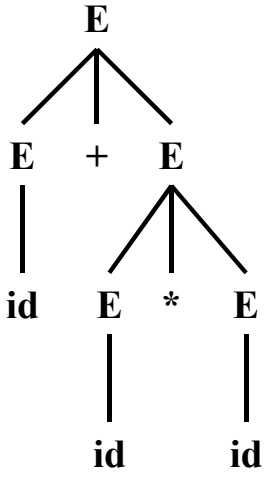
- **Ambiguous grammar:** A grammar produces more than one parse tree for a sentence

Grammar for Simple arithmetic expression

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

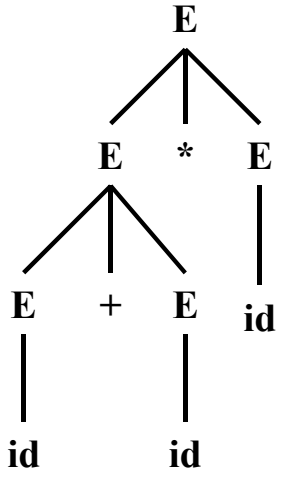
Derivation of id+id*id

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



Derivation of id+id*id

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

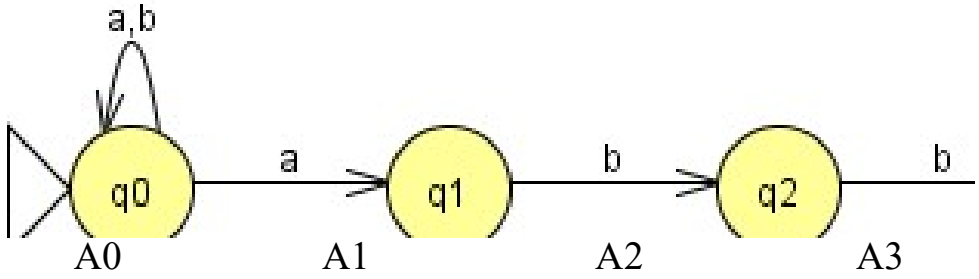


Writing a Grammar

- **Regular Expressions vs. Context Free Grammar**
  - ✓ Procedure to convert a nondeterministic finite automaton into a grammar
    - For each state i of the NFA, create a nonterminal symbol A_i
    - If state i has a transition to state j on symbol a, introduce the production $A_i \rightarrow aA_j$
    - If state I goes to state j on input Ɛ, introduce the production $A_i \rightarrow A_j$.
    - If I is an accepting state, introduce $A_i \rightarrow$ Ɛ.

Writing a Grammar

- **Regular Expressions vs. Context Free Grammar**
  - ✓ Procedure to convert a nondeterministic finite automaton into a grammar



$A0 \rightarrow aA0 \mid bA0 \mid aA1$

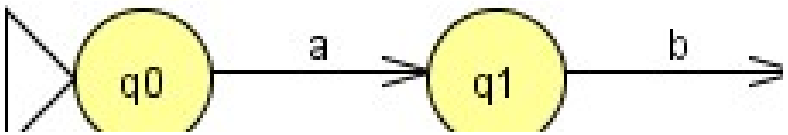
$A1 \rightarrow bA2$

$A2 \rightarrow bA3$

$A3 \rightarrow \varepsilon$

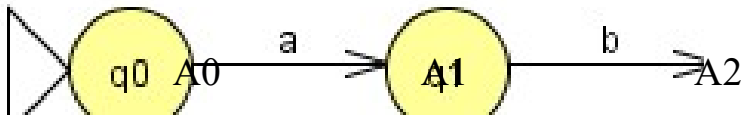
Think and Write

Write a grammar for the following nondeterministic finite automaton



Think and Write

Write a grammar for the following nondeterministic finite automaton



A0 → aA1
A1 → bA2
A2 → aA2 | bA2
A2 → ε

Writing a Grammar

- **Eliminating ambiguity**
  - ✓ For the most parsers, the grammar must be unambiguous.
  - ✓ Unambiguous grammar means unique selection of the parse tree for a sentence.
  - ✓ Consider following grammar

    stmt → if expr then stmt

    | if expr then stmt else stmt

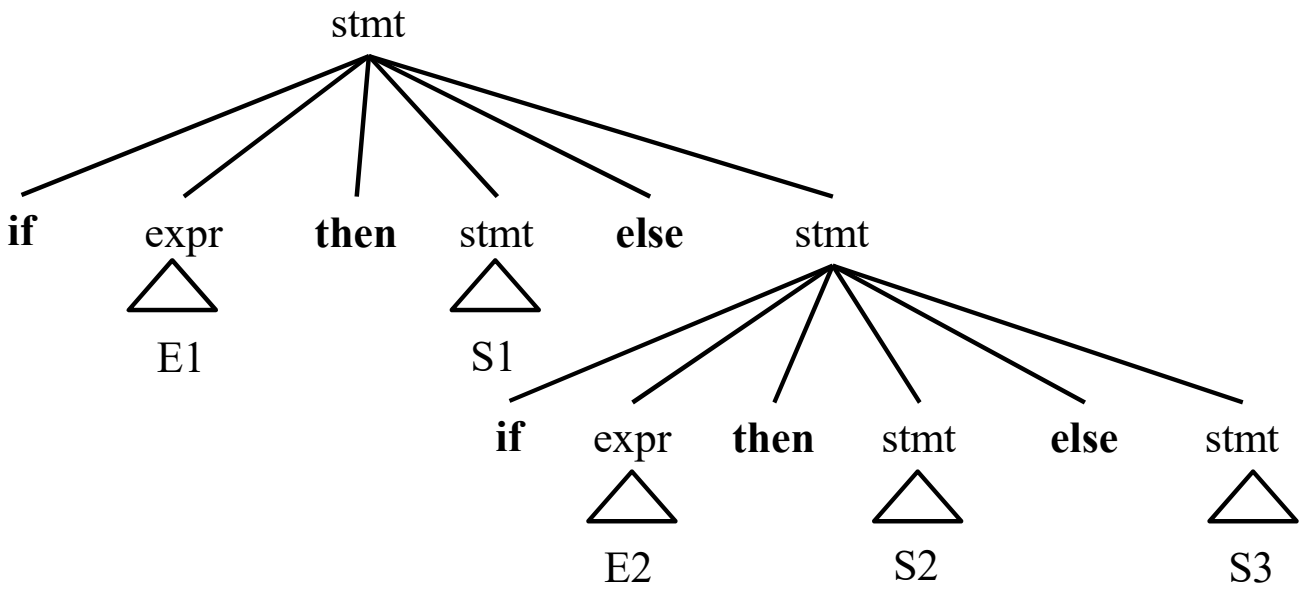
    | otherstmts

Writing a Grammar

- **Eliminating ambiguity**

  Consider example

  if E1 **then** S1 **else** if E2 **then** S2 **else** S3

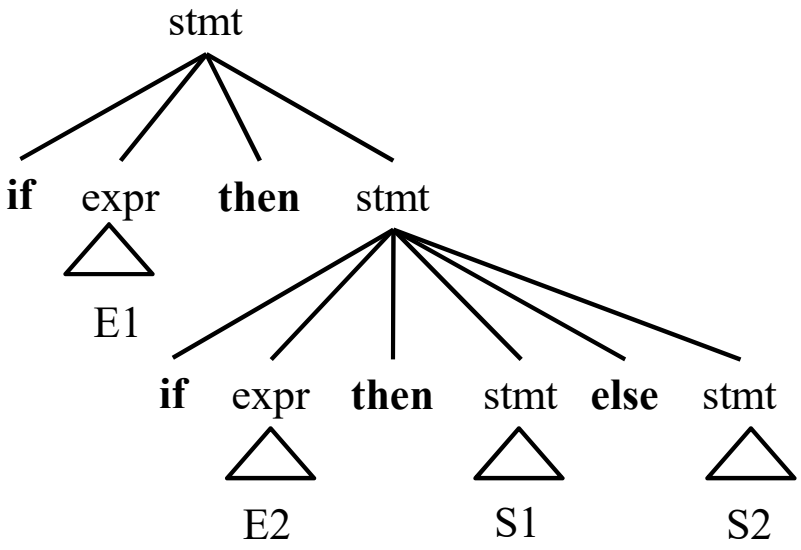


Writing a Grammar

- **Eliminating ambiguity**

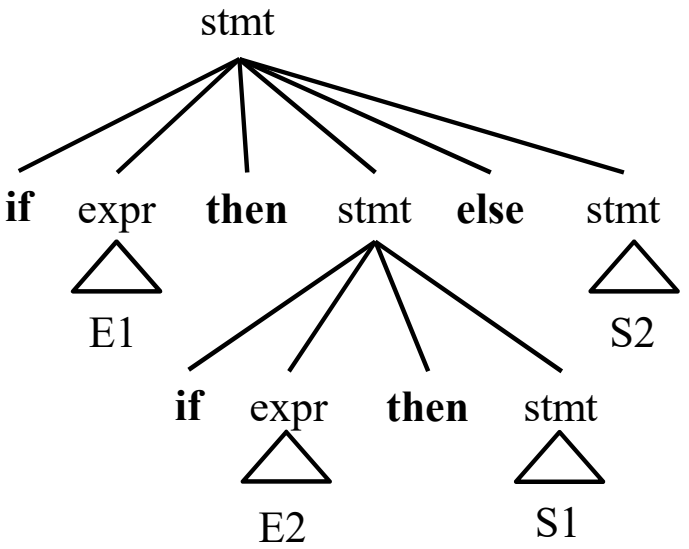
Consider example

if E1 **then** if E2 **then** S1 **else** S2



Correct parse tree

Rule: Match each ***else*** with the closest previous unmatched ***then***



Incorrect parse tree

Writing a Grammar

- **Eliminating ambiguity**
  - ✓ Grammar after eliminating ambiguity

The **idea** is that a statement appearing between a ***then*** and an ***else*** must be "matched" i.e., it must not end with an unmatched then followed by any statement, for the else would then be forced to match this unmatched then.

stmt → matchedstmt
|unmatchedstmt
matchedstmt → if expr then matchedstmt else matchedstmt
|otherstmts
unmatchedstmt → if expr then stmt
|if expr then matchedstmt else unmatchedstmt

Writing a Grammar

- **Eliminating ambiguity**

Ambiguous grammar	Grammar after eliminating ambiguity
E → E+E	E → E+T | T
| E*E	T → T*F | F
| E^E	F → G^F | G
| id	G → id | (E)
| (E)	

Writing a Grammar

- **Non Context Free Language Constructs**
  - ✓ Some languages can not be generated by any grammar.
  - ✓ Example: L1= { wcw| w is in (a |b)*} is not context-free.

    L2 = $\{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$ is not context free.

    L3 = $\{a^n b^n c^n \mid n \geq 0\}$ is not context free.

Writing a Grammar

- **Non Context Free Language Constructs**
  - ✓ But the languages similar to L1, L2 and L3 are context free.
  - ✓ Example:

L1' = { wcw^R| w is in (a |b)*} where w^R stands for reverse of w, is context-free

$S \rightarrow aSa \mid bSb \mid c$

L2' = {$a^n b^m c^m d^n$ | n≥1 and m≥1 } is context free.

$S \rightarrow aSd \mid aAd$

$A \rightarrow bAc \mid bc$

Writing a Grammar

- **Non Context Free Language Constructs**
  - ✓ But the languages similar to L1, L2 and L3 are context free.
  - ✓ Example:

L2'' = $\{a^n b^n c^m d^m \mid n \geq 1 \text{ and } m \geq 1\}$ is context free.

$S \rightarrow AB$

$A \rightarrow aAb \mid ab$

$B \rightarrow cBd \mid cd$

L3' = $\{a^n b^n \mid n \geq 1\}$ is context-free.

$S \rightarrow aSb \mid ab$

Writing a Grammar

- **Elimination of left recursion**
  - ✓ Left recursive grammar: A grammar is **left recursive if it has a non-terminal** A such that there is a derivation

$$A \overset{+}{\Rightarrow} A\alpha \text{ for some string } \alpha$$

Writing a Grammar

- **Elimination of left recursion**
  - ✓ Immediate left recursion:

    $A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

    ⇓ eliminate immediate left recursion

    $A \rightarrow \beta\,A'$

    $A' \rightarrow \alpha\,A' \mid \varepsilon$ an equivalent grammar

Writing a Grammar

- **Elimination of left recursion**
  - ✓ In general

    $A \rightarrow A\,\alpha 1 \mid A\,\alpha 2 \mid \ldots \mid A\,\alpha m \mid \beta 1 \mid \beta 2 \mid \ldots \mid \beta n$ where $\beta 1 \ldots \beta n$ do not start with A

    ⇓ eliminate immediate left recursion

    $A \rightarrow \beta 1\,A' \mid \beta 2\,A' \mid \ldots \mid \beta n\,A'$

    $A' \rightarrow \alpha 1\,A' \mid \alpha 2\,A' \mid \ldots \mid \alpha m\,A' \mid \varepsilon$ an equivalent grammar

Writing a Grammar

- **Elimination of left recursion**
  - ✓ Immediate left recursion

    Example

    $E \rightarrow E+T \mid T$

    $T \rightarrow T*F \mid F$

    $F \rightarrow id \mid (E)$

    ⇓ eliminate immediate left recursion

    $E \rightarrow T\ E'$

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

    $T \rightarrow F\ T'$

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

    $F \rightarrow id \mid (E)$

Writing a Grammar

- **Elimination of left recursion**

✓ A grammar cannot be immediately left-recursive, but it still can be left-recursive

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$

This grammar is not immediately left-recursive, but it is still left-recursive.

$S \Rightarrow Aa \Rightarrow Sca$ or

$A \Rightarrow Sc \Rightarrow Aac$ causes to a left-recursion

✓ So, we have to eliminate all left-recursions from our grammar

Writing a Grammar

- **Elimination of left recursion**
  - ✓ Eliminating left recursion algorithm

Input: Grammar G with no cycles or ε-productions

Output: An equivalent grammar with no left recursion

Method:

- Arrange non-terminals in some order: A1 ... An
- **for** i:= 1 **to** n **do begin**
  - **for** j:=1 **to** i-1 **do begin**

replace each production of the form $A_i \rightarrow A_j \gamma$

by the productions $Ai \rightarrow \alpha_1 \gamma \mid ... \mid \alpha_k \gamma$

where $Aj \rightarrow \alpha_1 \mid ... \mid \alpha_k$

end

- eliminate immediate left-recursions among Ai productions

end

Writing a Grammar

- **Elimination of left recursion**
  - ✓ Eliminate left recursion algorithm

Example

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: S, A

for S:

-we do not enter the inner loop as there is no immediate left recursion in S.

for A:

- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$
- So $A \rightarrow Ac \mid Aad \mid bd$
- after eliminating the immediate left-recursion in A

$A \rightarrow bdA' \mid fA'$

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

An equivalent grammar which is **not left-recursive** is:

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid fA'$

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

Think and Write

Eliminate left recursion from the following grammar productions

$S \rightarrow Sab \mid Scd \mid e$

Think and Write

Eliminate left recursion from the following grammar productions

$S \rightarrow Sab \mid Scd \mid e$

After eliminating left recursion from grammar

$S \rightarrow eS'$

$S' \rightarrow abS' \mid cdS' \mid \varepsilon$

Writing a Grammar

- **Left factoring**
  - ✓ Useful for producing a grammar suitable for predictive parsing.

stmt → if expr then stmt else stmt

|if expr then stmt

when we see ***if***, we cannot immediately tell which production rule to choose to rewrite ***stmt*** in the derivation.

Writing a Grammar

- **Left factoring**
  - ✓ In general, $A \rightarrow \alpha\beta1 \mid \alpha\beta2$ where α is non-empty and the first symbols of $\beta1$ and $\beta2$ are different.
  - ✓ When processing α, we cannot know whether expand A to $\alpha\beta1$ or A to $\alpha\beta2$
  - ✓ But, if we re-write the grammar as follows

    $A \rightarrow \alpha A'$

    $A' \rightarrow \beta1 \mid \beta2$

    so, we can immediately expand A to $\alpha A'$

Writing a Grammar

- **Left factoring**
  - ✓ For each non-terminal A with two or more production rules with a common nonempty prefix

    $A \rightarrow \alpha\beta1 \mid ... \mid \alpha\beta n \mid \gamma 1 \mid ... \mid \gamma m$

    convert it into

    $A \rightarrow \alpha A' \mid \gamma 1 \ ... \mid \gamma m$

    $A' \rightarrow \beta 1 \mid ... \mid \beta n$

Writing a Grammar

- **Left factoring**
  - ✓ Example 1:

stmt → if expr then stmt else stmt

|if expr then stmt

S → iEtS | iEtSeS | a
E → b

⇓

S → iEtSS' | a
S' → eS | ε
E → b

Writing a Grammar

- **Left factoring**
  - ✓ Example 2:

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid g \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid \underline{cdg} \mid \underline{cde}B \mid \underline{cdf}B$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Top-Down Parsing

- The parse tree is created top to bottom.
- Top-down parser
  - ✓ **Recursive decent parser**
  - ✓ **Predictive parser**
  - ✓ **Nonrecursive predictive parser**

Top-Down Parsing

- **Recursive decent parser**
  - ✓ Backtracking is needed
  - ✓ It tries to find the left-left most derivation.
  - ✓ Example:

    S → cAd

    A → ab | a

    Input string w = cad

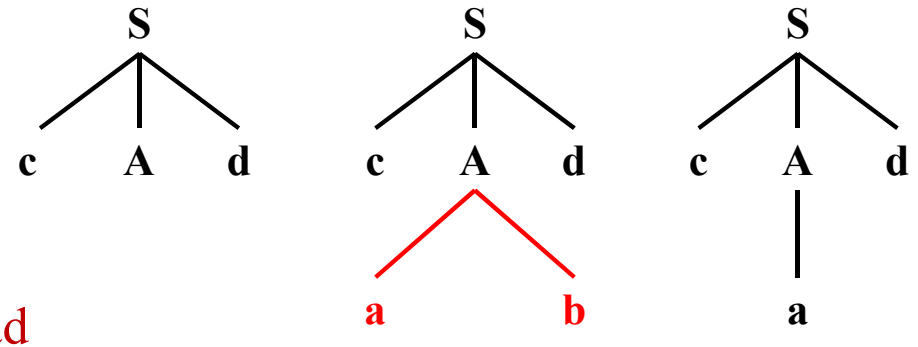


Fig: a Fig: b Fig: c

Fails, backtrack

Think and Write

Construct a recursive-descent parser with backtracking for the grammar:

$S \rightarrow aSbS \mid bSaS \mid \varepsilon$ and input string w = abab

Think and Write

Construct a recursive-descent parser with backtracking for the grammar:

S → aSbS | bSaS | Ɛ and input string w = abab

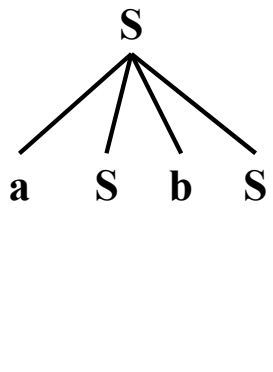
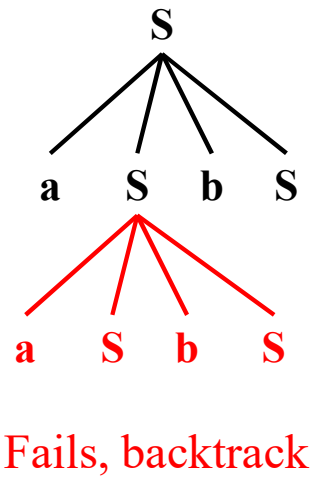


Fig: a



Fails, backtrack

Fig: b

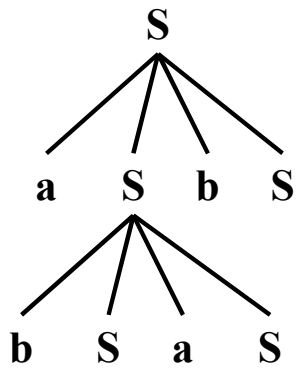


Fig: c

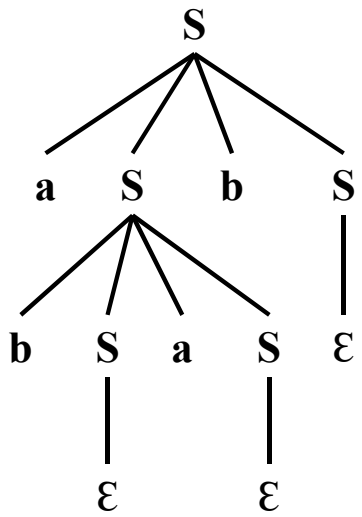


Fig: d

Top-Down Parsing

- **Predictive Parser**
  - ✓ Need no backtracking if
    - ➢ Carefully write a grammar
    - ➢ Eliminate left recursion from grammar
    - ➢ Left factoring the resulting grammar
  - ✓ Example

    stmt → **if** *expr* **then** *stmt* **else** *stmt*

    | **while** *expr* **do** *stmt*

    | **begin** *stmt_list* **end**

Top-Down Parsing

- **Predictive Parser**
  - ✓ Transition diagram for predictive parser : To construct the transition diagram of a predictive parser from a grammar
    - ➢ Eliminate left recursion from the grammar
    - ➢ Left factor the grammar
    - ➢ For each nonterminal A do the following:
      - ▪ Create an initial and final state
      - ▪ For each production A → X1X2...Xn, create a path from initial top the final state with edges labeled X1, X2, ... Nn.

Top-Down Parsing

- **Predictive Parser**
  - ✓ Predictive parser behaves as follows:
    - ➢ It begins in the start state from the start symbol.
    - ➢ If it is in state *s* with an edge labeled by terminal *a* to state *t* and if the next input symbol is *a* then the parser moves the input cursor one position right and goes to state *t*.
    - ➢ If edge is labeled by a nonterminal *A* then parser goes to the start state for *A* without moving the input cursor.
    - ➢ If there is an edge from *s* to *t* labeled $\mathcal{E}$, then from state s the parser immediately goes to state without advancing the input.

Top-Down Parsing

- **Predictive Parser**
  - ✓ Example: **E → T E', E' → +T E' | ε,T → F T', T' → *F T' | ε, F → id | (E)**

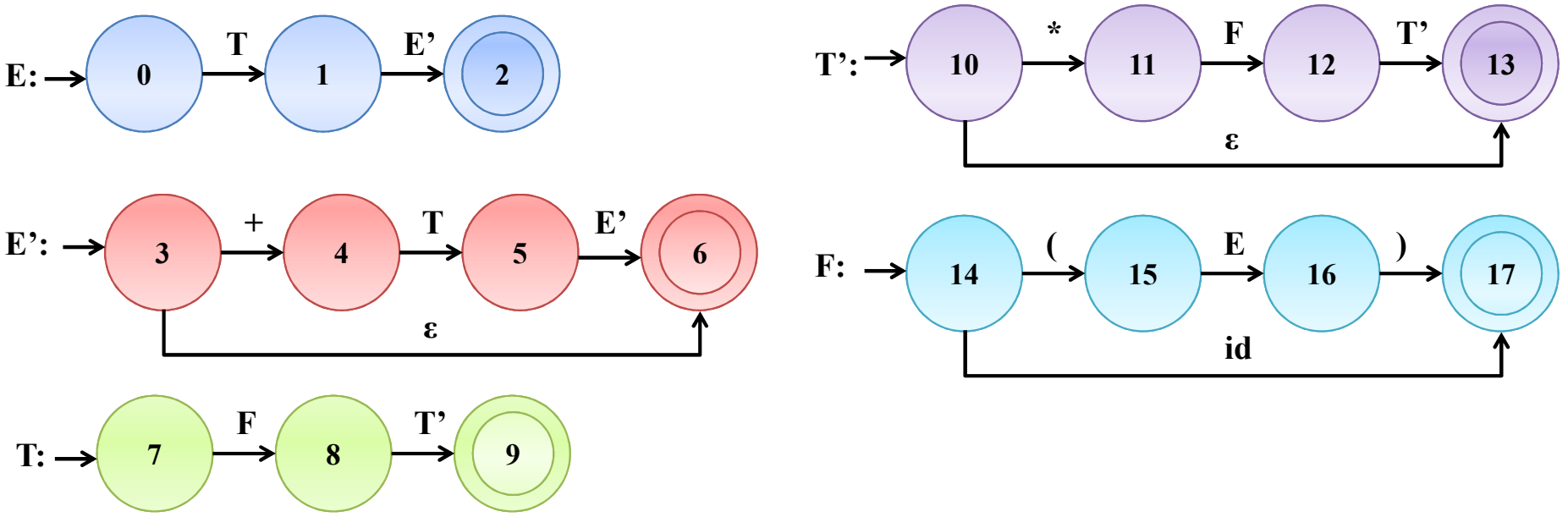
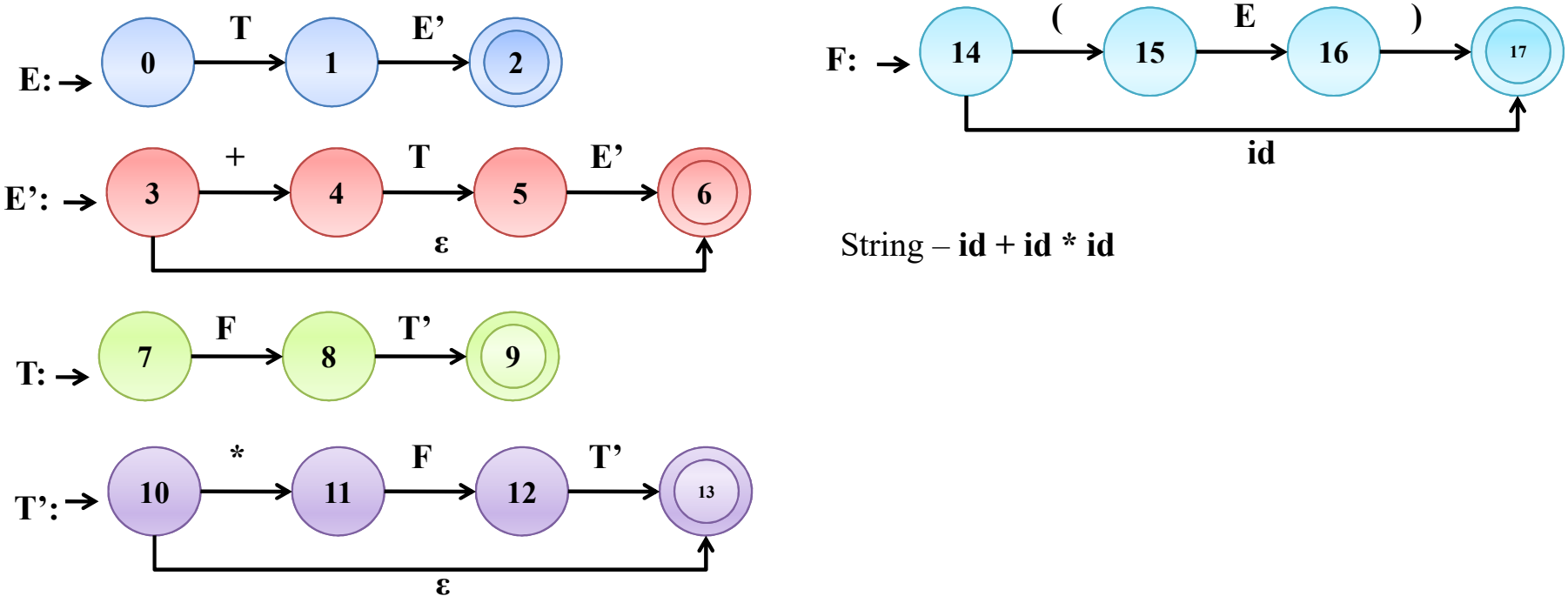


Figure: Transition diagram for grammar

Top-Down Parsing

- **Predictive Parser**



String – **id + id * id**

Figure: Transition diagram for grammar

Top-Down Parsing

- **Predictive Parser**

✓ Example: **E → T E' E' → +T E' | ε T → F T' T' → *F T' | ε F → id | (E)**

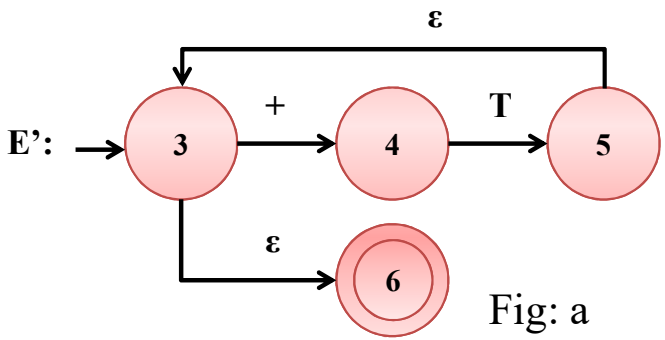


Fig: a

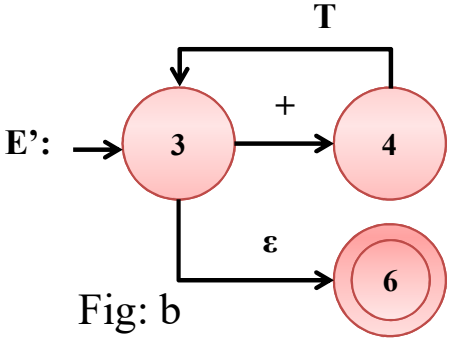
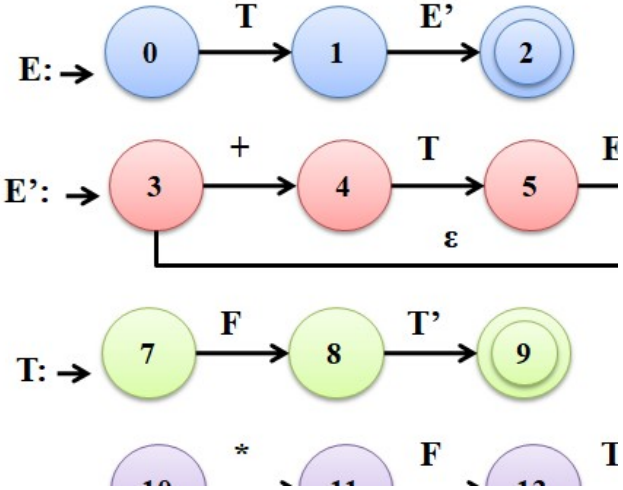


Fig: b



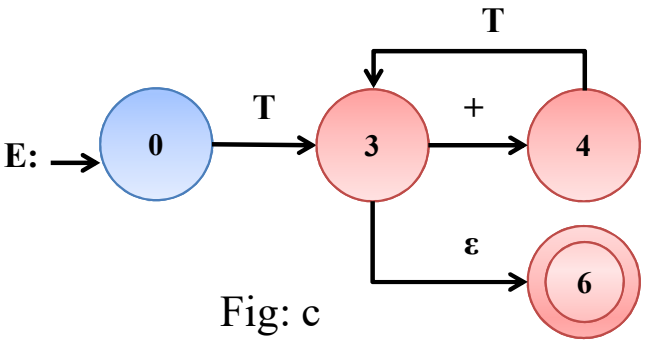


Fig: c

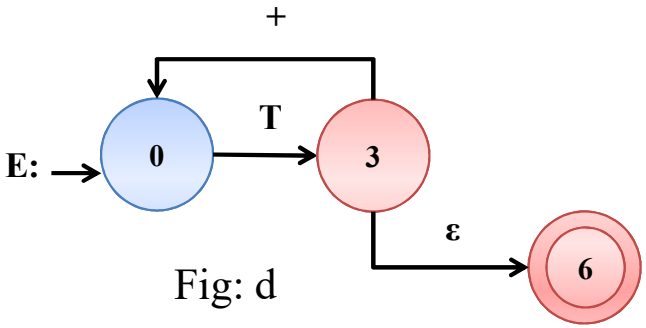


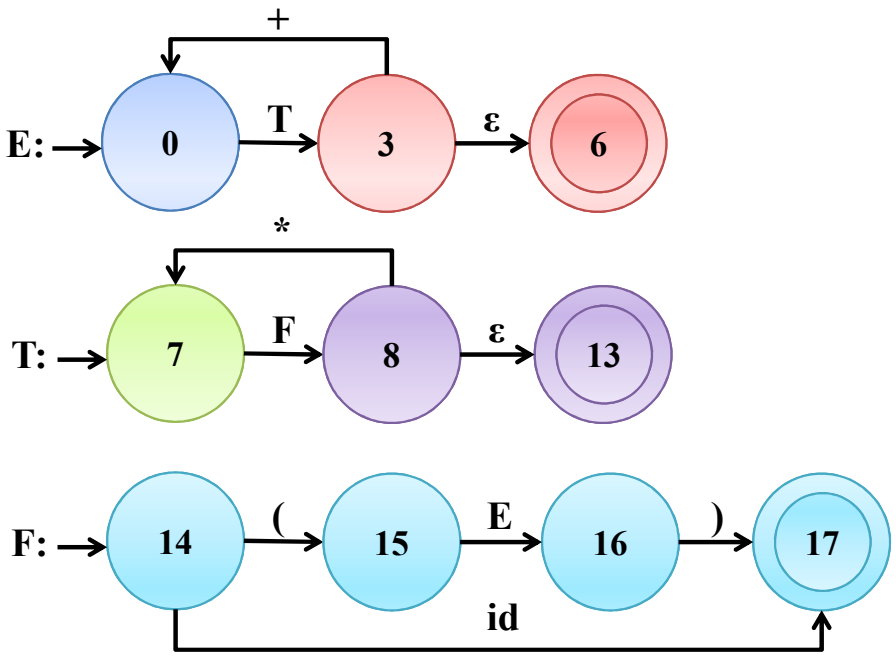
Fig: d

Figure: Simplified transition diagram for grammar

Top-Down Parsing

- **Predictive Parser**

✓ Example: **E → T E' E' → +T E' | ε T → F T' T' → *F T' | ε F → id | (E)**



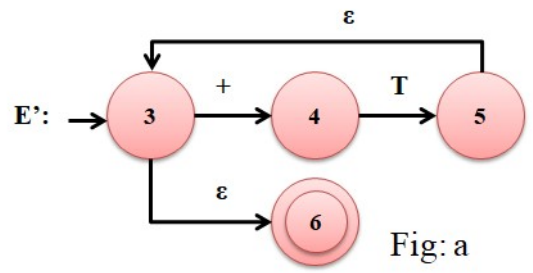
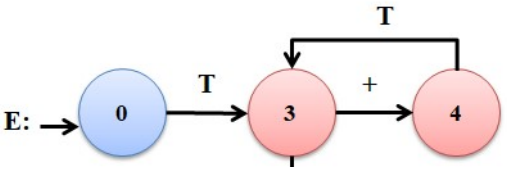


Fig: a



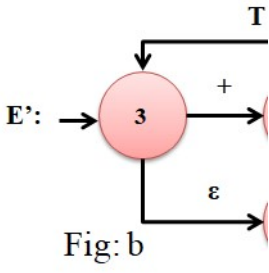


Fig: b

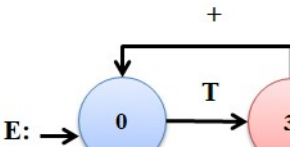
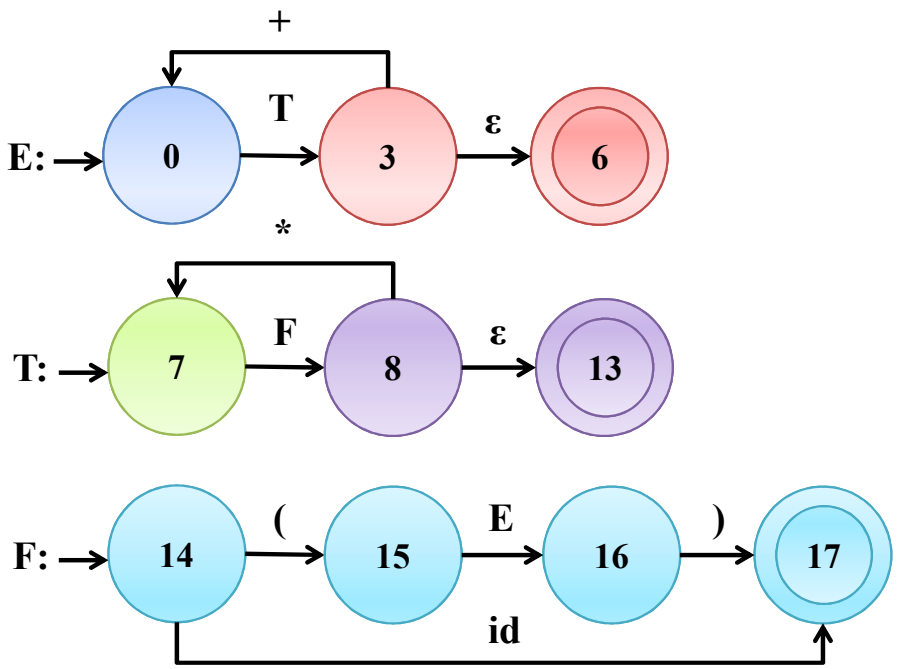


Figure: Simplified transition diagram for grammar

Top-Down Parsing

- **Predictive Parser**

✓ Example: **E → T E' E' → +T E' | ε T → F T' T' → *F T' | ε F → id | (E)**



Input string: id+id

Figure: Simplified transition diagram for grammar

Top-Down Parsing

- The parse tree is created top to bottom.
- Top-down parser
  - ✓ **Recursive decent parser**
  - ✓ **Predictive parse**
  - ✓ **Nonrecursive predictive parser**

Top-Down Parsing

- **Nonrecursive predictive parser**

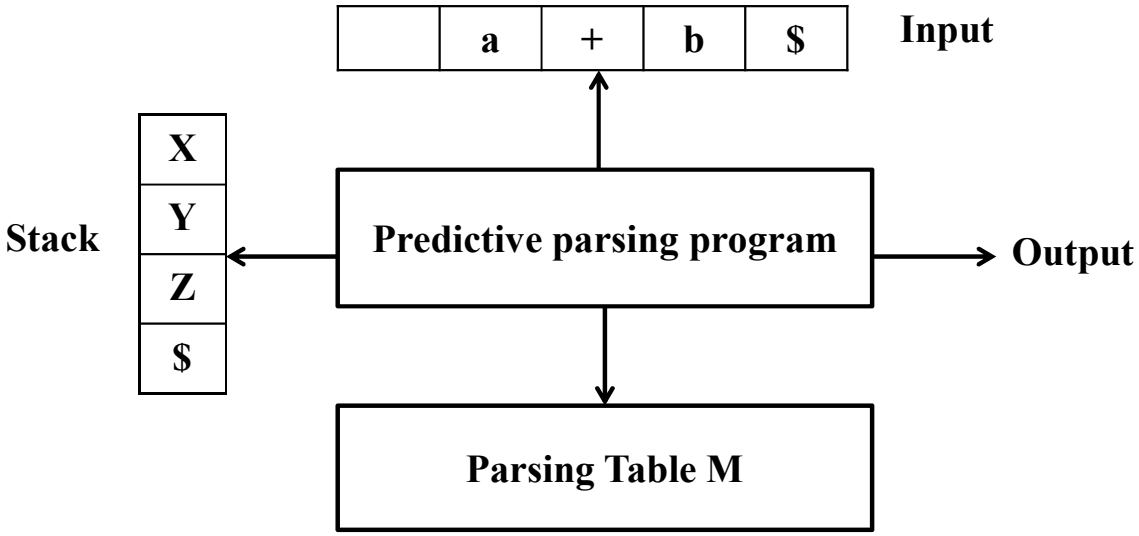


Figure 1: Model of non-recursive predictive parser

Top-Down Parsing

- **Nonrecursive predictive parser**

  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.

  1. If X = a = $ then parser halts (successful completion)
  2. If X and a are the same terminal symbol (different from $)
     - parser pops X from the stack, and moves the next symbol in the input buffer.

Top-Down Parsing

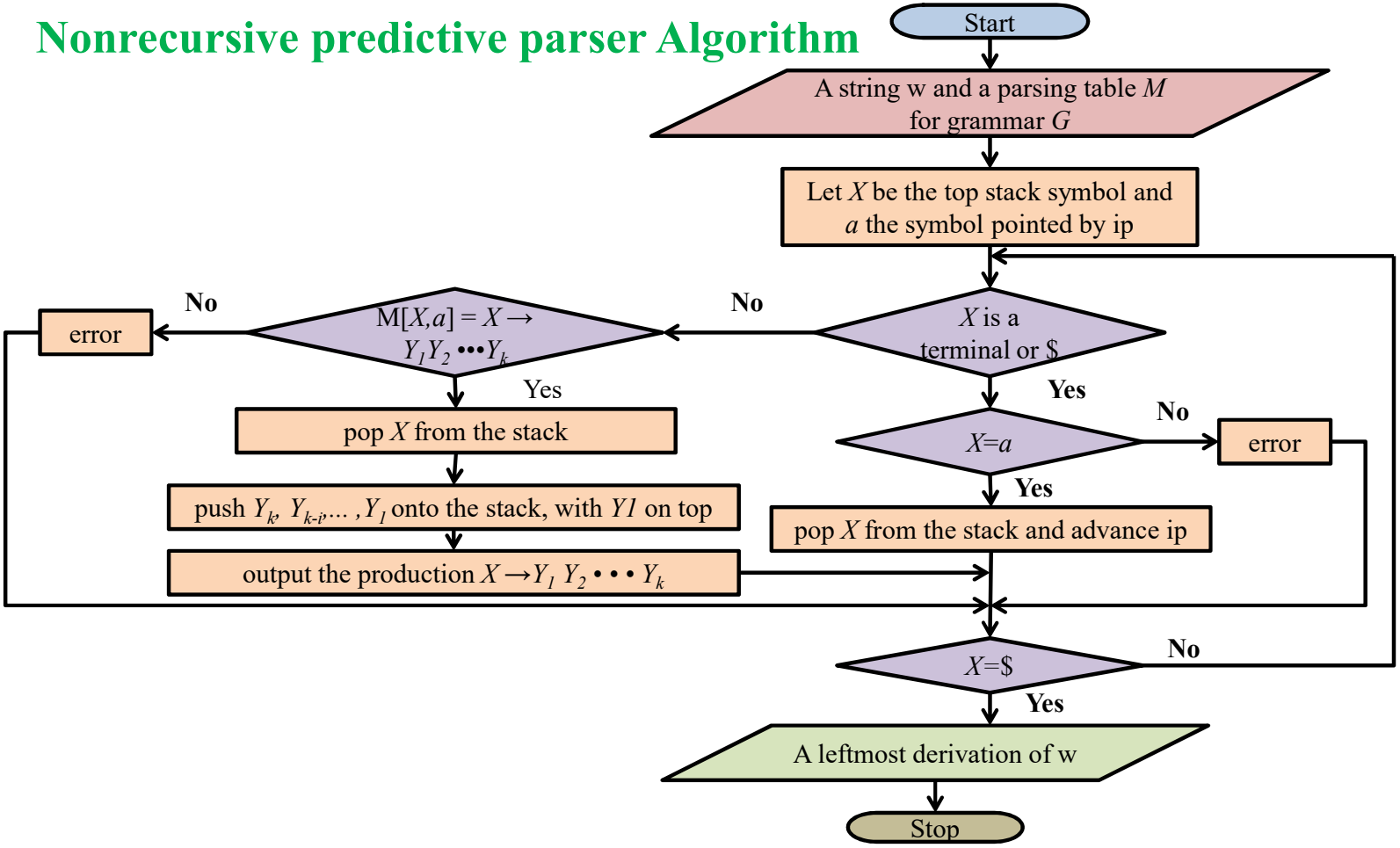
- **Nonrecursive predictive parser**

3. 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.

4. none of the above then 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.

Top-Down Parsing

- **Nonrecursive predictive parser Algorithm**



Think and Write

What is difference between recursive descent parsing, predictive parsing and non-recursive predictive parsing?

Think and Write

What is difference between recursive descent parsing, predictive parsing and non-recursive predictive parsing?

Recursive descent parsing – requires backtracking .

Predictive parsing – requires implicit recursive call.

Non-recursive predictive parsing – maintain a stack.

Top-Down Parsing

- **Nonrecursive predictive parser**

Example 1:

$E \rightarrow T\ E'$

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

$T \rightarrow F\ T'$

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

$F \rightarrow id \mid (E)$

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

Input String : id + id

Top-Down Parsing

- **Nonrecursive predictive parser**

stack	input	output
$E	id+id$	E → TE’
$E’T	id+id$	T → FT’
$E’ T’F	id+id$	F → id
$ E’ T’id	id+id$	
$ E’ T’	+id$	T’ → ε
$ E’	+id$	E’ → +TE’
$ E’ T+	+id$	
$ E’ T	id$	T → FT’
$ E’ T’ F	id$	F → id
$ E’ T’id	id$	
$ E’ T’	$	T’ → ε
$ E’	$	E’ → ε
$	$	accept

	id	+	*	(	
E	$E \rightarrow TE'$			$E \rightarrow TE'$	
E’		$E' \rightarrow +TE'$			$E' -$
T	$T \rightarrow FT'$			$T \rightarrow FT'$	
T’		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' -$

Top-Down Parsing

- **Nonrecursive predictive parser**

  Example 2:

  S → aBa

  B → bB | ε

  Input String : abba

	a	b	
S	S → aBa		
B	B → ε	B → bB	

stack	input	output	
$S	abba$	S → aBa	--------1
$aBa	abba$		--------2
$aB	bba$	B → bB	--------3
$aBb	bba$		--------4
$aB	ba$	B → bB	--------5
$aBb	ba$		--------6
$aB	a$	B → ε	--------7
$a	a$		--------8
$	$	Accept	--------9

FIRST Set

- If α is any string of grammar symbol then FIRST(α) is the set of terminals that begin the string derived from α.
- To compute FIRST(α) for all grammar symbols X, apply following rules until no more terminals or ε to any FIRST set
  - ✓ **Rule 1:** If X is terminal then FIRST(X) is $\{X\}$.
  - ✓ **Rule 2:** If $X \rightarrow \varepsilon$ is a production then add **ε** to FIRST(X).
  - ✓ **Rule 3:** If X is nonterminal and $X \rightarrow Y_1Y_2\ldots Y_k$ is a production then place a in FIRST(X) if for some i, a is in FIRST(Yi) and ε is in all of FIRST(Y_1),…, FIRST(Y_{i-1}) i.e. $Y_1\ldots Y_{i-1} \rightarrow^* \varepsilon$. If ε is in FIRST($Y_j$) for $j=1,2,\ldots,k$ then add ε to FIRST(X).

FIRST Set

- Example

  E→TE’

  E’ →+TE’|ε

  T →FT’

  T’ →*FT’|ε

  F →(E)|id

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

  FIRST(E’)={+, ε}

  FIRST(T’)={*, ε}

FIRST Set

- Example

  S→ iEtSS' | a

  S' →eS | ε

  E → b

  FIRST(S)={i, a}

  FIRST(S')={e, ε}

  FIRST(E)={b}

FIRST Set

- Example

  S→ AaAb | BbBa

  A → ε

  B → ε

  FIRST(S)={a, b}

  FIRST(A)={ε}

  FIRST(B)={ε}

Think and Write

Compute FIRST set for the following grammar:

S→ CC

C → cC| d

Think and Write

Compute FIRST set for the following grammar:

S→ CC

C → cC| d

FIRST(S) = {c, d}

FIRST(C) = {c, d}

FOLLOW Set

- For nonterminal A, FOLLOW(A) is to be the set of terminals a that can appear immediately to the right of A in some sentential form.
- To compute FOLLOW(A) for all nonterminal A, apply following rules until nothing can be added to any FOLLOW set
  - ✓ **Rule 1:** Place $\$$ in FOLLOW(S), where S is the start symbol and $\$$ is the input right endmarker
  - ✓ **Rule 2:** If there is production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) except for ε is placed in FOLLOW(B)
  - ✓ **Rule 3:** If there is production $A \rightarrow \alpha B$ or production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε then everything in FOLLOW(A) is in FOLLOW(B).

FOLLOW Set

- Example

E→TE’

E’ →+TE’

E’ →ε

T →FT’

T’ →*FT’

T’ →ε

F →(E)

F →id

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

FIRST(E’)={+, ε}

FIRST(T’)={*, ε}

FOLLOW(E)=FOLLOW(E’)={), $}

FOLLOW(T)=FOLLOW(T’)={+,), $}

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

FOLLOW Set

- Example

S→ iEtSS’ **FIRST(S)={i, a}**

S→ a **FIRST(S’)={e, ε}**

S’ → eS **FIRST(E)={b}**

S’ → ε

E → b **FOLLOW(S)={e, $}**

FOLLOW(S’)={e, $}

FOLLOW(E)={t}

FOLLOW Set

- Example

S→ AaAb

S→ BbBa

A → ε

B → ε

FIRST(S)={a, b}

FIRST(A)={ε}

FIRST(B)={ε}

FOLLOW(S)={$}

FOLLOW(A)={a, b}

FOLLOW(B)={a, b}

Think and Write

Compute FOLLOW set for the following grammar:

S→ CC

C → cC| d

Think and Write

Compute FIRST set for the following grammar:

S→ CC

C → cC

C → d

FIRST(S) = {c, d}

FIRST(C) = {c, d}

FOLLOW(S) = {$}

FOLLOW(C) = {c, d, $}

Construction of Predictive Parsing Table

- Algorithm: Construction of a predictive parsing table

  Input: Grammar G

  Output: Parsing Table M

  Method:

  1) For each production $A \rightarrow \alpha$ of the grammar, do step 2 and 3.

  2) For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.

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

  4) Make each undefined entry of M to be *error* .

Construction of Predictive Parsing Table

- Example

  E→TE’

  E’ →+TE’|ε

  T →FT’

  T’ →*FT’|ε

  F →(E)|id

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

FIRST(E’)={+, ε}

FIRST(T’)={*, ε}

FOLLOW(E)=FOLLOW(E’)={), $}

FOLLOW(T)=FOLLOW(T’)={+,), $}

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

Construction of Predictive Parsing Table

E→TE’

E’ →+TE’

E’ → ε

T →FT’

T’ →*FT’

T’ → ε

F →(E)|id

FIRST(E) = {(, id}

FIRST(T) = {(, id}

FIRST(F) = {(, id}

FIRST(E’)={+, ε}

FIRST(T’)={*, ε}

FOLLOW(E) = {), $}

FOLLOW(E’) = {), $}

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

FOLLOW(T’) = {+,), $}

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

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)		

Construction of Predictive Parsing Table

- Example

  S→ iEtSS' | a

  S' →eS | ε

  E → b

FIRST(S)={i, a} **FOLLOW(S)={e, $}**

FIRST(S')={e, ε} **FOLLOW(S')={e, $}**

FIRST(E)={b} **FOLLOW(E)={t}**

Construction of Predictive Parsing Table

FIRST(S)={i, a}

FIRST(S')={e, ε}

FIRST(E)={b}

FOLLOW(S)={e, $}

FOLLOW(S')={e, $}

FOLLOW(E)={t}

S→ iEtSS'

S→ a

S' →eS

S' → ε

E → b

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

Construction of Predictive Parsing Table

- Example

  S→ AaAb | BbBa

  A → ε

  B → ε

FIRST(S)={a, b} **FOLLOW(S)={$}**

FIRST(A)={ε} **FOLLOW(A)={a, b}**

FIRST(B)={ε} **FOLLOW(B)={a, b}**

Construction of Predictive Parsing Table

S→ AaAb

S→ BbBa

A → ε

B → ε

FIRST(S)={a, b}

FIRST(A)={ε}

FIRST(B)={ε}

FOLLOW(S)={$}

FOLLOW(A)={a, b}

FOLLOW(B)={a, b}

Non-terminal	Input Symbol		
	a	b	$
S	S→ AaAb	S→ BbBa	
A	A → ε	A → ε	
B	B → ε	B → ε	

Think and Write

Construct predictive parsing table for the following grammar:

S→ CC

C → cC| d

Think and Write

Construct predictive parsing table for the following grammar:

S→ CC

C → cC| d

FIRST(S)={c, d}

FIRST(C)={c, d}

FOLLOW(S)={$}

FOLLOW(C)={c, d, $}

S→ CC

C → cC

C → d

Non-terminal	Input Symbol		
	c	d	$
S	S→ CC	S→ CC	
C	C → cC	C → d	

LL(1) Grammar

- A grammar whose parsing table has no multiply-defined entries.

This grammar is LL(1).

E→TE’

E’ →+TE’|ε

T →FT’

T’ →*FT’|ε

F →(E)|id

FIRST(E)=FIRST(T)=FIRST(F)={(, id}	FOLLOW(E)=FOLLOW(E’)={), $}
FIRST(E’)={+, ε}	FOLLOW(T)=FOLLOW(T’)={+,), $}
FIRST(T’)={*, ε}	FOLLOW(F)={+, *,), $}

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)		

LL(1) Grammar

This grammar is not LL(1).

S→ iEtSS’ | a

S’ →eS | ε

E → b

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

Multiply-defined entries

Non-terminal	Input Symbol					
	a	b	e	i	t	$
S	S→ a			S→ iEtSS’		
S’			S’ → ε S’ →eS			S’ →ε
E		E → b				

LL(1) Grammar

- A grammar G is LL(1) iff whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - ✓ **Condition 1**: For no terminal a, do both α and β derive strings beginning with a.

    (FIRST(α) ∩ FIRST(β) = ϕ)
  - ✓ **Condition 2**: At most one of α and β can derive empty string.
  - ✓ **Condition 3**: If $\beta \xrightarrow{*} \varepsilon$ then α does not derive any string beginning with a terminal in FOLLOW(A).

    (FIRST(α) ∩ FOLLOW(A) = ϕ)

LL(1) Grammar

This grammar is LL(1).

S→ AaAb | BbBa

A → ε

B → ε

FIRST(S)={a, b}

FIRST(A)={ε}

FIRST(B)={ε}

FOLLOW(S)={$}

FOLLOW(A)={a, b}

FOLLOW(B)={a, b}

FIRST(AaAb) ∩ FIRST(BbBa) = ϕ

{a} ∩ {b} = ϕ

Non-terminal	Input Symbol		
	a	b	$
S	S→ AaAb	S→ BbBa	
A	A → ε	A → ε	
B	B → ε	B → ε	

LL(1) Grammar

This grammar is not LL(1).

S→ iEtSS' | a

S' →eS | ε

E → b

FIRST(S)={i, a}	FOLLOW(S)={e, $}
FIRST(S')={e, ε}	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' → ε S' →eS			S' →ε
E		E → b				

Multiply-defined entries

1) FIRST(iEtSS') ∩ FIRST(a) = ϕ
{i} ∩ {a} = ϕ

2) FIRST(eS) ∩ FIRST(ε) = ϕ
{e} ∩ {ε} = ϕ

3) FIRST(eS) ∩ FOLLOW(S') = ϕ
{e} ∩{e, $}≠ ϕ

Think and Write

Check following grammar for LL(1):

$S \rightarrow CC$

$C \rightarrow cC \mid d$

Think and Write

Check following grammar for LL(1):

S→ CC

C → cC| d

This grammar is LL(1).

S→ CC

C → cC| d

FIRST(S)={c, d}	FOLLOW(S)={$}
FIRST(C)={c, d}	FOLLOW(C)={c, d, $}

FIRST(cC) ∩ FIRST(d) = ϕ

{c} ∩ {d} = ϕ

Non-terminal	Input Symbol		
	c	d	$
S	S→ CC	S→ CC	
C	C → cC	C → d	

Error Recovery for Predictive Parsing

- An error is detected during predictive parsing when
  - ✓ The terminal on the top of the stack does not match the next input symbol
  - ✓ When nonterminal ***A*** is on top of the stack, ***a*** is the next input symbol and the parsing table entry ***M[A, a]*** is empty.

Error Recovery for Predictive Parsing

- Error Recovery
  - ✓ **Panic mode recovery**
  - ✓ **Phrase level recovery**

Error Recovery for Predictive Parsing

- Error Recovery
  - ✓ **Panic mode recovery**: skip symbols on the input until a token in a selected set of synchronizing tokens appear.
    - ➢ Use **FOLLOW** symbols as synchronizing tokens.
    - ➢ Use ***"synch"*** in predictive parsing table to indicate synchronizing tokens obtains from the FOLLOW set of the nonterminal.

Error Recovery for Predictive Parsing

- Error Recovery
  - ✓ **Panic mode recovery**:

    Rules:

    1. If parser looks up entry ***M[A, a]*** and finds it blank then the input symbol ***a*** is skipped.
    2. If the entry is ***synch***, then the nonterminal on top of the stack is popped in an attempt to resume parsing.
    3. If a token on the top of the stack does not match the input symbol, then we pop the token from the stack

Error Recovery for Predictive Parsing

FIRST(E)=FIRST(T)=FIRST(F)={(, id}	FOLLOW(E)=FOLLOW(E')={), $}
FIRST(E')={+, ε}	FOLLOW(T)=FOLLOW(T')={+,), $}
FIRST(T')={*, ε}	FOLLOW(F)={+, *,), $}

E→TE'

E' →+TE'|ε

T →FT'

T' →*FT'|ε

F →(E)|id

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	E→TE'			E→TE'	synch	synch
E'		E' →+TE'			E' →ε	E' →ε
T	T →FT'	synch		T →FT'	synch	synch
T'		T' →ε	T' →*FT'		T' →ε	T' →ε
F	F →id	synch	synch	F →(E)	synch	synch

Error Recovery for Predictive Parsing

Sr. No.	Stack	Input	Remark
1	$E	)id*+id$	Error, skip)
2	$E	id*+id$	id is in FIRST(E)
3	$E’T	id*+id$	
4	$E’T’F	id*+id$	
5	$E’T’id	id*+id$	
6	$E’T’	*+id$	
7	$E’T’F*	*+id$	
8	$E’T’F	+id$	Error, M[F, +] = synch
9	$E’T’	+id$	F has been popped
10	$E’	+id$	
11	$E’T+	+id$	
12	$E’T	id$	
13	$E’T’F	id$	
14	$E’T’id	id$	
15	$E’T’	$	
16	$E’	$	
17	$	$	

$E \rightarrow TE'$ $\quad E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$ $\quad T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) | id$

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	E→TE’			E→TE’	synch	synch
E’		E’ →+TE’			E’ →ε	E’ →ε
T	T →FT’	synch		T →FT’	synch	synch
T’		T’ →ε	T’ →*FT’		T’ →ε	T’ →ε
F	F →id	synch	synch	F →(E)	synch	synch

Error Recovery for Predictive Parsing

- Error Recovery
  - ✓ **Phrase level recovery**: implemented by filling in the blank entries in the predictive parsing table with pointers to error routines.

Shift-reduce Parsing

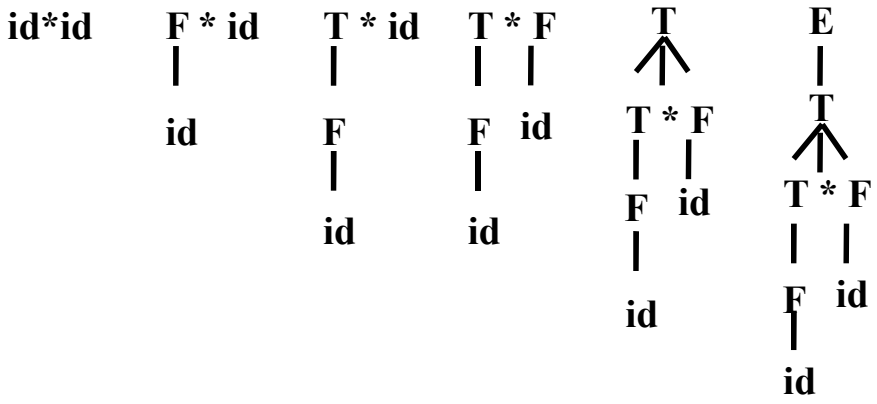
- Shift-reduce parsing constructs a parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)

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

String: id*id

Derivation: id*id
F*id
T*id
T*F
T
E

Parse Tree



Handles

- 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.

Example:

E→E+E	E → E*E
E →(E)	E → id

E → <u>E + E</u>
→E + <u>E * E</u>
→E + E * <u>id3</u>
→E + <u>id2</u> * id3
→<u>id1</u>+ id2 * id3

E → <u>E * E</u>
→E * <u>id3</u>
→E + E * <u>id3</u>
→E + <u>id2</u> * id3
→<u>id1</u> + id2 * id3

Handle Pruning

- Rightmost derivation in reverse can be obtained by handle pruning.

Example

E→E+E
E → E*E
E →(E) | id

Right sentential form	Handle	Reducing production
id1+id2*id3	id1	E → id
E+id2*id3	id2	E → id
E+E*id3	id3	E → id
E+E*E	E*E	E → E*E
E*E	E+E	E→E+E
E		

Right sentential form	Handle	Reducing production
id1+id2*id3	id1	E → id
E+id2*id3	id2	E → id
E+E*id3	id3	E → id
E+E*E	E*E	E → E*E
E+E	E+E	E→E+E
E		

Handle Pruning

- Rightmost derivation in reverse can be obtained by handle pruning.

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

Right sentential form	Handle	Reducing production
id1 +id2*id3	id	F → id
F+ id2*id3	F	T → F
T+ id2* id3	T	E → T
E+id2*id3	id2	F → id
E+F * id3	F	T → F
E+T*id3	id3	F → id
E+T*F	T*F	T → T*F
E+T	E+T	E→E+T
E		

Stack Implementation of 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 $

Stack Implementation of Shift Reduce Parsing

- Basic operations:
  - ✓ Shift
  - ✓ Reduce
  - ✓ Accept
  - ✓ Error

Stack Implementation of Shift Reduce Parsing

- Example

E→E+E	E → E*E
E →(E)	E → id

Sr. No.	Stack	Input	Action
1	$	id1+id2*id3$	Shift
2	$id1	+id2*id3$	Reduce by E → id
3	$E	+id2*id3$	Shift
4	$E+	id2*id3$	Shift
5	$E+id2	*id3$	Reduce by E → id
6	$E+E	*id3$	Shift
7	$E+E*	id3$	Shift
8	$E+E*id3	$	Reduce by E → id
9	$E+E*E	$	Reduce by E → E*E
10	$E+E	$	Reduce by E → E+E
11	$E	$	Accept

Sr. No.	Stack	Input	Action
1	$	id1+id2*id3$	Shift
2	$id1	+id2*id3$	Reduce by E → id
3	$E	+id2*id3$	Shift
4	$E+	id2*id3$	Shift
5	$E+id2	*id3$	Reduce by E → id
6	$E+E	*id3$	Shift
7	$E+E*	id3$	Shift
8	$E+E*id3	$	Reduce by E → id
9	$E+E*E	$	Reduce by E → E+E
10	$E*E	$	Reduce by E → E*E
11	$E	$	Accept

Think and Write

Derive the string id+id from following grammar using shift reduce parsing

E→E+T | T	T→ T*F | F
E →(E) | id	

Think and Write

Derive the following string using shift reduce parsing

E→E+T | T T→ T*F | F
E →(E) | id

Sr. No.	Stack	Input	Action
1	$	id+id$	Shift
2	$id	+id$	Reduce by F → id
3	$F	+id$	Reduce by T → F
4	$T	+id$	Reduce by E → T
5	$E	+id$	Shift
6	$E+	id$	Shift
7	$E+id	$	Reduce by F → id
8	$E+F	$	Reduce by T → F
9	$E+T	$	Reduce by E → E+T
10	$E	$	Accept

Conflict during Shift Reduce Parsing

- Two kind of conflicts
  - ✓ Shift/reduce conflict
  - ✓ Reduce/reduce conflict

Conflict during Shift Reduce Parsing

- Shift/reduce conflict

  Example:

  stmt→ **if** *expr* **then** *stmt*

  | **if** *expr* **then** *stmt* **else** *stmt*

  | **other**

Stack	Input
…**if** *expr* **then** *stmt*	*else*…$

Conflict during Shift Reduce Parsing

- Reduce/reduce conflict

Example:

(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*

Statement A(I, J)

Stack	**Input**
… id(id	,id) …$

Operator Grammar

- Operator grammar
  - ✓ No production right side is ε.
  - ✓ No production has to adjacent nonterminal.

    Example:

    E → E op E | id

    op → + | *

    The above grammar is not an operator grammar but:

    E → E + E | E* E | id

Operator Precedence Parsing

- a $\gtrdot$ b means that terminal "a" has the higher precedence than terminal "b".
- a $\lessdot$ b means that terminal "a" has the lower precedence than terminal "b".
- a $\doteq$ b means that the terminal "a" and "b" both have same precedence.
- **Note:**
  - ✓ id has higher precedence than any other symbol
  - ✓ $ has lowest precedence.

Operator Precedence Parsing

- **Operator precedence relations from Associativity and precedence**
  - ✓ If operator θ1 has higher precedence than operator θ2then make θ1 ⋗ θ2 and θ2 ⋖ θ1.

    e.g. * ⋗ + and + ⋖ *
  - ✓ If θ1 and θ2 are operators of equal precedence then make θ1 ⋗ θ2 and θ2 ⋗ θ1 if the operators are left associative or make θ1 ⋖ θ2 and θ2 ⋖ θ1 if they are right associative.

    e.g. + ⋗ +, + ⋗ - Left Associative

    ↑ ⋖ ↑ Right Associative

Operator Precedence Parsing

- **Operator precedence relations from Associativity and precedence**
  - ✓ Make

    $\theta \lessdot id$, $id \gtrdot \theta$, $\theta \lessdot ($, $(\lessdot \theta$,

    $) \gtrdot \theta$, $\theta \gtrdot)$, $\theta \gtrdot \$$ and $\$ \lessdot \theta$ for all operators θ.

    Also

    $(\doteq)$ $\$ \lessdot ($ $\$ \lessdot id$

    $(\lessdot ($ $id \gtrdot \$$ $) \gtrdot \$$

    $(\lessdot id$ $id \gtrdot)$ $) \gtrdot)$

Operator Precedence Parsing

- **Operator precedence relations from Associativity and precedence**
  - ✓ ↑ is of highest precedence and right associative
  - ✓ * and / are of next highest precedence and left associative
  - ✓ + and – are of lowest precedence and left associative.

Operator Precedence Parsing

- **Using Operator precedence relations**

  Example: E→ E+E | E*E | id

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	

Operator Precedence Parsing

- **Process to find the handle**
  - ✓ Both end of the given input string, add the $ symbol.
  - ✓ Now scan the input string from left to 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

- **Process to find the handle**
  - ✓ In input string $\$a_1a_2...a_n\$$, insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).
  - ✓ Example: E→ E+E | E*E | id

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

  - ✓ Then the input string id+id*id with the precedence relations inserted will be:

$$\$ <\cdot \text{ id } \cdot> + <\cdot \text{ id } \cdot> * <\cdot \text{ id } \cdot> \$$$

Operator Precedence Parsing

- **Parsing action**

Example: E→ E+E | E*E | id

String: id+id*id

	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

$ id+id*id $
$ id ⋗+id*id $
$ ⋖ id ⋗+id*id $
$ E+ id ⋗ *id $
$ E+ ⋖ id ⋗ *id $
$ E+ E*id ⋗ $
$ E+ E * ⋖ id ⋗ $
$ E+ E * E$
$ ⋖ + E * E$
$ ⋖ + ⋖ * E$
$ ⋖ + ⋖ * ⋗ $
$ ⋖ + ⋗ $
$ $

Operator Precedence Parsing

Algorithm: Operator precedence parsing algorithm

Input: An input string w and a table of precedence relations

Output: If w is well formed, a skeletal parse tree with a placeholder nonterminal E labeling all interior nodes otherwise error indication

Method: Initially, the stack contains $ and the input buffer, the string *w*$

Operator Precedence Parsing

Set *ip* to point to the first symbol of *w*$;
repeat forever
    if $ is on top of the stack and *ip* points to $ **then**
        return
    else begin
        let *a* be the topmost terminal symbol on the stack
        and let *b* be the symbol pointed to by ip;
        if $a \lessdot b$ or $a \doteq b$ **then begin**
            push *b* onto the stack;
            advance *ip* to the next input symbol;
        end;
        else if $a \gtrdot b$ **then**
            repeat
                pop the stack
            until the top stack terminal is related by $\lessdot$
            to the terminal most recently popped
        else *error()*
    end

Operator Precedence Parsing

- **Parsing action**

Example: E→ E+E | E*E | id

String: id+id*id

	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

Sr. No.	STACK	INPUT	ACTION
1	$	id + id * id$	$ ⋖ id, push
2	$ id	+ id * id$	id ⋗ +, pop
3	$	+ id * id$	$ ⋖ +, push
4	$ +	id * id$	+ ⋖ id, push
5	$ + id	* id$	id ⋗ *, pop
6	$ +	* id$	+ ⋖ *, push
7	$ + *	id$	* ⋖ id, push
8	$ + * id	$	id ⋗ $, pop
9	$ + *	$	* ⋗ $, pop
10	$ +	$	+ ⋗ $, pop
11	$	$	Accept

Think and Write

Parse the string using operator precedence parsing using following table of precedence relations.

String: (id+id)*id

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	≐	
)	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<		

Think and Write

String: (id+id)*id

Sr. No.	STACK	INPUT	ACTION
1	$	(id+id)*id$	$ ⋖ id, push
2	$(	id+id)*id$	(⋖ id, push
3	$(id	+id)*id$	id ⋗ +, pop
4	$(+	id)*id$	+ ⋖ id, push
5	$(+id	)*id$	id ⋗), pop
6	$(+	)*id$	+ ⋗), pop
7	$(	)*id$	(≐), pop
8	$	*id$	$ ⋖ *, push
9	$*	id$	* ⋖ id, push
10	$*id	$	id ⋗ $, pop
11	$*	$	* ⋗ $, pop
12	$	$	Accept

	+	-	*	/	↑	id	(	)	$
+	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
-	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
/	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
↑	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
id	⋗	⋗	⋗	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	⋖	⋖	≐	
)	⋗	⋗	⋗	⋗	⋗			⋗	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	⋖		

Handling Unary Operators

- Let ¬ to be a unary prefix operator then we make
  - **θ** ⋖ ¬ for any operator
  - ¬ ⋗ **θ** if ¬ has higher precedence than **θ**
  - ¬ ⋖ **θ** if ¬ has lower (or equal) precedence than **θ**

  Example – if ¬ has higher precedence than & and & is left associative then group the expression E& ¬E&E as (E&(¬E))&E by above rules

Handling Unary Operators

- Operator-Precedence parsing cannot handle the unary minus when we also have the binary minus in our grammar.
- The best approach to solve this problem, let the lexical analyzer handle this problem.
  - The lexical analyzer will return two different tokens for the unary minus and the binary minus.
  - The lexical analyzer will need a lookhead to distinguish the binary minus from the unary minus.

Parsing Function

- Compilers using operator precedence parser need not store the table of precedence table.
- The table can be encoded by two precedence functions f and g that map terminal symbols to integers.
- For symbol a and b
  1. $f(a) < g(b)$ whenever $a \lessdot b$
  2. $f(a) = g(b)$ whenever $a \doteq b$ and
  3. $f(a) > g(b)$ whenever $a \gtrdot b$

Parsing Function

- **Algorithm – Constructing precedence functions**

  Input – An operator precedence matrix

  Output – Precedence function representing the input matrix or an indication that none exists.

  Method –

  1. Create symbols f_a and g_a for each a that is a terminal or $.
  2. Partition the created symbols into as many group as possible in such a way that if a $\doteq$ b then f_a and g_b are in the same group. We may have to put the symbols in the same group even if they are not related by $\doteq$.

Parsing Function

- **Algorithm – Constructing precedence functions**

  Input – An operator precedence matrix

  Output – Precedence function representing the input matrix or an indication that none exists.

  Method –

  3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of gb to the group of fa if a $\lessdot$ b, otherwise if a $\gtrdot$ b place an edge from the group of fa to that of gb.
  4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of fa and gb respectively.

Parsing Function

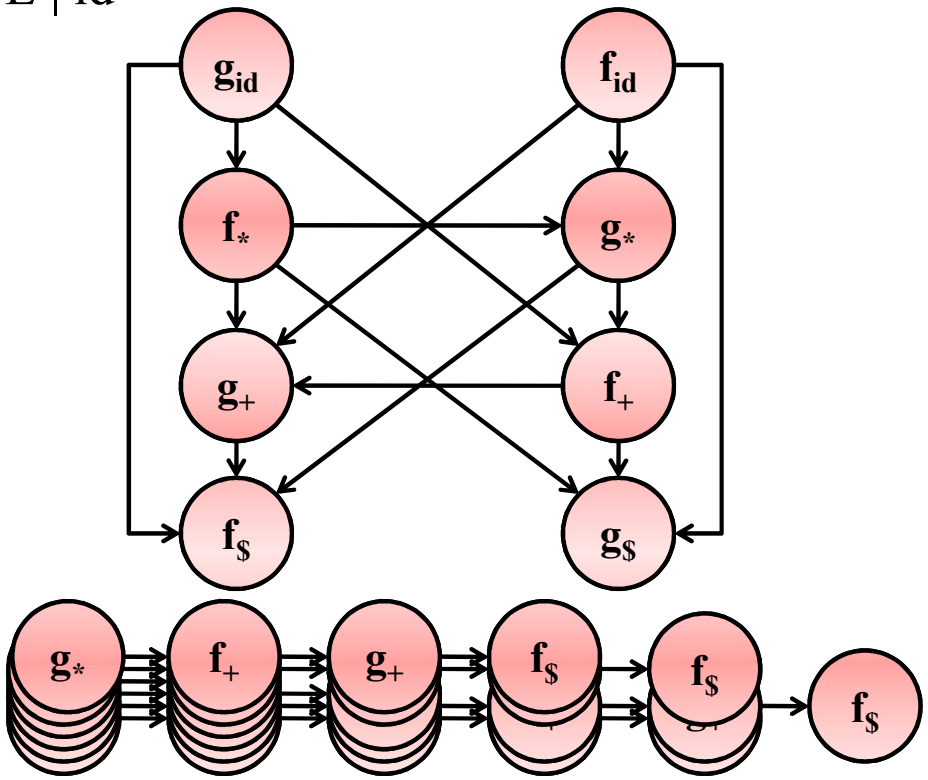
- **Algorithm – Constructing precedence functions**

  Example: E→ E+E | E*E | id

g

f	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

	id	+	*	$
f	4	2	4	0
g	5	1	3	0



Think and Write

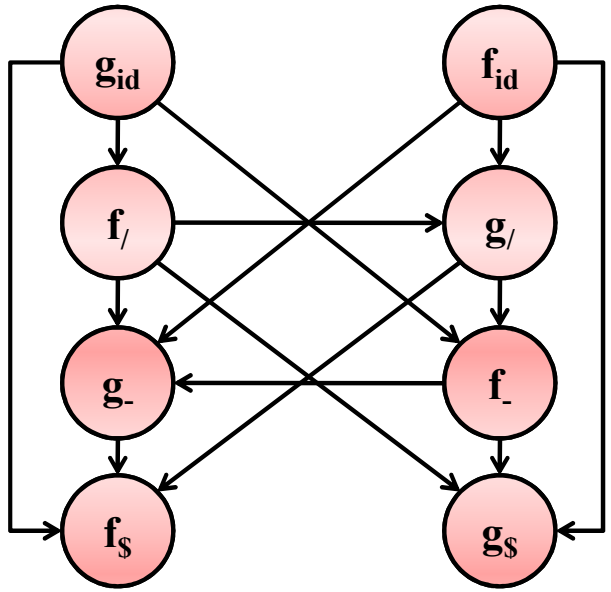
Find the precedence function for following operator precedence relations

	id	-	/	$
id		$\dot{>}$	$\dot{>}$	$\dot{>}$
-	$\dot{<}$	$\dot{>}$	$\dot{<}$	$\dot{>}$
/	$\dot{<}$	$\dot{>}$	$\dot{>}$	$\dot{>}$
$	$\dot{<}$	$\dot{<}$	$\dot{<}$	

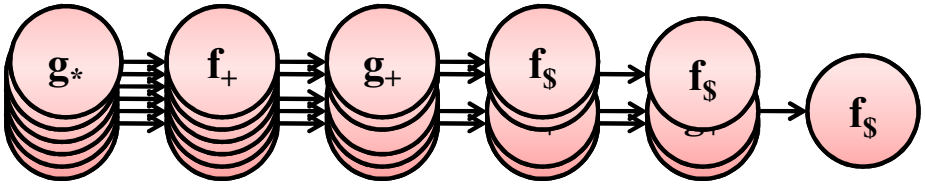
Pause the video now and answer the questions

Think and Write

f \ g	id	-	/	$
id		>	>	>
-	<	>	<	>
/	<	>	>	>
$	<	<	<	



	id	+	*	$
f	4	2	4	0
g	5	1	3	0



Error Recovery in Operator Precedence Parsing

- There are two points in the parsing process at which an operator-precedence parser can discover **syntactic errors**:
  1. **No relation holds between the terminal on the top of stack and the next input symbol.**
  2. **A handle is found (reduction step), but there is no production with this handle as a right side**
- **Error Recovery:**
  - ✓ Each empty entry is filled with a pointer to an error routine.
  - ✓ Decides the popped handle "looks like" which right hand side. And tries to recover from that situation.

Error Recovery in Operator Precedence Parsing

- **Handling Errors during Reduction**

> **2. A handle is found (reduction step), but there is no production with this handle as a right side**

- ✓ For errors of type 2, as there is no production to reduce by, so a diagnostic message is printed instead.
- ✓ To determine what the diagnostic should say, the routine handling case (2) must decide what production the right side being popped "**looks like.**"

Error Recovery in Operator Precedence Parsing

- **Handling Errors during Reduction**
  - ✓ Example - Suppose abc is popped, and there is no production right side consisting of a,b,c together with zero or more nonterminals.
    - ➢ Then we might consider if deletion of one of a, b and c yields a legal right side (nonterminals omitted).
    - ➢ For example, if there were a right side aEcE, we might issue the diagnostic:

      illegal b on line (line containing b)

Error Recovery in Operator Precedence Parsing

- **Handling Errors during Reduction**
  - ✓ Example - Suppose abc is popped, and there is no production right side consisting of a,b,c together with zero or more nonterminals.
    - ➢ We might also consider changing or inserting a terminal. Thus if abEdc. were a right side, we might issue a diagnostic:
    - ➢ **missing d on line (line containing c)**
    - ➢ We may also find that there is a right side with the proper sequence of terminals, but the wrong pattern of nonterminal. If abc is poped off the stack with no intervening w surrounding nonterminals, and abc is not a right side but aEbc k is, we might issue a diagnostic:
    - ➢ **missing E on line (line containing b)**

Error Recovery in Operator Precedence Parsing

- **Handling Errors during Reduction**
  - ✓ In general, the difficulty of determining appropriate diagnostics when no legal right side is found depends upon whether there are a finite or infinite number of possible strings
  - ✓ Any such string b1b2 ...bk , must have $\doteq$ relations holding between adjacent symbols, so b1 $\doteq$ b2 $\doteq$...$\doteq$ bk.

Error Recovery in Operator Precedence Parsing

- **Handling Errors during Reduction**
  - However, in order for a path b1b2 ...bk to be "poppable" on some Input:
    - ✓ There must be a symbol a (possibly $) such that

      $a \lessdot b_1 = b_2 = \ldots = b_k$

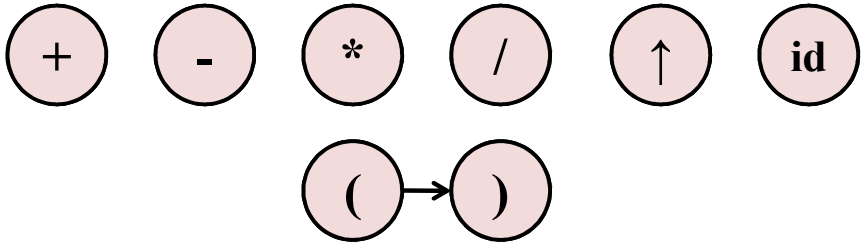
    - ✓ There must be a symbol c (possibly $) such that $b_k \gtrdot c$

    only then could a reduction be called for and $b_1 b_2 \ldots b_k$ be the sequence of symbols popped.

Error Recovery in Operator Precedence Parsing

- **Handling Errors during Reduction**
  - If the graph has a path from an initial to a final node containing a cycle, then there are an infinity of strings that might be popped; otherwise, there are only a finite number.
  - Grammar E->E+E | E-E | E*E | E/E | E↑E | (E) | -E | id

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	≐	
)	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<		



Error Recovery in Operator Precedence Parsing

- **Handling Errors during Reduction**
  - Specifically, the checker does the following:
    - ✓ If +,-,*,/ or ↑, is reduced, it checks that nonterminals appear on both sides. If not, it issues the diagnostic: **missing operand**
    - ✓ If id is reduced, it checks that there is no nonterminal to the right or left. If there is, it can warn: **missing operand**
    - ✓ If ( ) is reduced, it checks that there is a nonterminal between the parentheses, if not, it can say **no expression between parentheses**
    - ✓ Also it must check that no nonterminnl appears on either side of the parentheses. If one does, it issues the same diagnostic: **missing operand**

Error Recovery in Operator Precedence Parsing

- **Handling Shift/Reduce Error**
  - When consulting the precedence matrix to decide whether to shift or reduce, we may find that no relation holds between the top stack and the first input symbol.
  - To recover, we must modify (insert/change)
    - ✓ Stack or
    - ✓ Input or
    - ✓ Both.
  - We must be careful that we don't get into an infinite loop.

Error Recovery in Operator Precedence Parsing

- **Handling Shift/Reduce Error**
  - ✓ Example – Consider the precedence matrix.

	id	(	)	$
id	e3	e3	⋖	⋗
(	⋖	⋖	≐	e4
)	e3	e3	⋗	⋗
$	⋖	⋖	e2	e1

Error Recovery in Operator Precedence Parsing

- **Handling Shift/Reduce Error**
  - ✓ Example – Consider the precedence matrix.

	id	(	)	$
id	e3	e3	⋖	⋗
(	⋖	⋖	≐	e4
)	e3	e3	⋗	⋗
$	⋖	⋖	e2	e1

  - ➢ e1: Called when : whole expression is missing
    insert id onto the input
    issue diagnostic: '**missing operand**'
  - ➢ e2: Called when : expression begins with a right parenthesis
    delete) from the input
    issue diagnostic: '**unbalanced right parenthesis**'

Error Recovery in Operator Precedence Parsing

- **Handling Shift/Reduce Error**
  - ✓ Example – Consider the precedence matrix.

	id	(	)	$
id	e3	e3	⋖	⋗
(	⋖	⋖	≐	e4
)	e3	e3	⋗	⋗
$	⋖	⋖	e2	e1

  - ➢ e3: Called when : id or) is followed by id or (

    insert + onto the input

    issue diagnostic: '**missing operator**'
  - ➢ e4: Called when : expression ends with a left parenthesis

    pop (from the stack

    issue diagnostic: '**missing right parenthesis**'

Think and Write

Identify the errors in the following operator precedence relation table

	+	-	*	/	id	$
+	⋗	⋗	⋖	⋖	⋖	⋗
-	⋗	⋗	⋖	⋖	⋖	⋗
*	⋗	⋗	⋗	⋗	⋖	⋗
/	⋗	⋗	⋗	⋗	⋖	⋗
id	⋗	⋗	⋗	⋗		⋗
$	⋖	⋖	⋖	⋖	⋖	

Think and Write

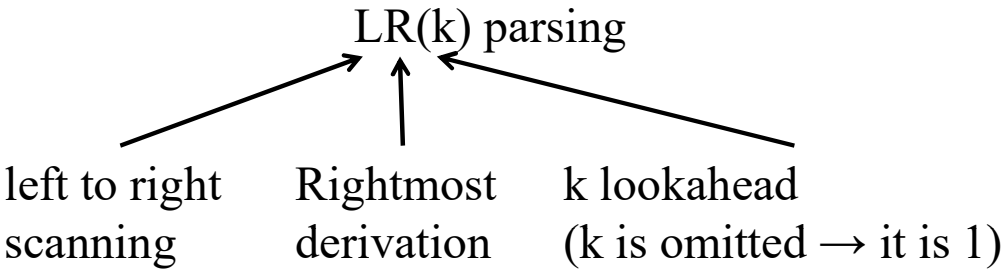
	+	-	*	/	id	$
+	>	>	<	<	<	>
-	>	>	<	<	<	>
*	>	>	>	>	<	>
/	>	>	>	>	<	>
id	>	>	>	>	e3	>
$	<	<	<	<	<	e1

- ✓ e1: Called when : whole expression is missing
  insert id onto the input
  issue diagnostic: '**missing operand**'

- ✓ e3: Called when : id or) is followed by id or (
  insert + onto the input
  issue diagnostic: '**missing operator**'

LR Parser

- Bottom-up technique to parse a large class of context-free-grammars



LR Parser

- LR parsing is attractive because:
  - ✓ LR parser can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
  - ✓ LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - ✓ The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

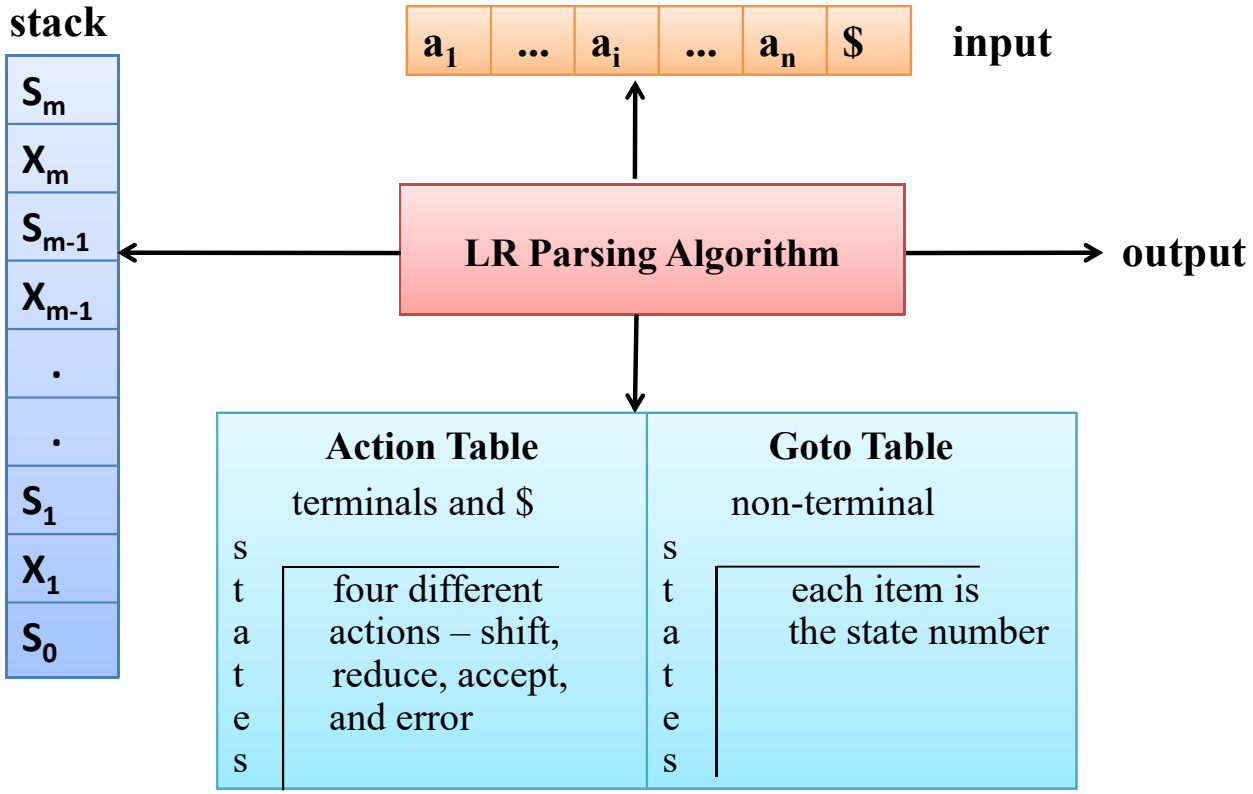
    LL(1)-Grammars ⊂ LR(1)-Grammars
  - ✓ An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

LR Parser

- It covers wide range of grammars.
  - ✓ **SLR** – simple LR parser
  - ✓ **LR** – most general LR parser
  - ✓ **LALR** – intermediate LR parser (look-head LR parser)
- SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

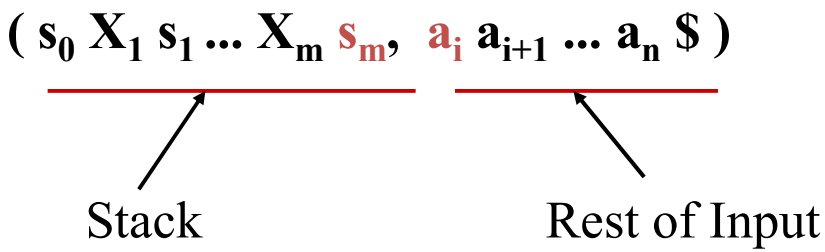
LR Parsing Algorithm

- **LR Parsing Algorithm**



LR Parsing Algorithm

- **LR Parsing Algorithm**
  - ✓ A configuration of a LR parsing is:



s_m and a_i decides the parser action by consulting the parsing action table.

  - ✓ A configuration of a LR parsing represents the right sentential form:

X_1 ... X_m a_i a_{i+1} ... a_n $

LR Parsing Algorithm

- **LR Parsing Actions**
  - ✓ **shift s** - shifts the next input symbol and the state **s** onto the stack
    ($s_o X_1 s_1 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$$) ➔ ($s_o X_1 s_1 \ldots X_m s_m \; a_i \; s, a_{i+1} \ldots a_n \$$)
  - ✓ **reduce A→β**
    pop $2|\beta|$ items from the stack;
    then push **A** and **s** where **s=goto[s_{m-r},A]**

    ($s_o X_1 s_1 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$$) ➔ ($s_o X_1 s_1 \ldots X_{m-r} \; s_{m-r} \; A \; s, a_i \ldots a_n \$$)

    Output is the reducing production reduce A→β
  - ✓ **Accept** – Parsing successfully completed
  - ✓ **Error** -- Parser detected an error (an empty entry in the action table)

LR Parsing Algorithm

- **Algorithm – LR parsing algorithm**

  Input – An input string w and an LR parsing table with function action and goto for a grammar G

  Output – If w is in L(G), a bottom up parse for w otherwise an error indication.

  Method – Initially a parser has s_0 on its stack where s0 is the initial state and w$ in the input buffer. The parser then executes the program until an accept or error action is encountered.

LR Parsing Algorithm

```
set ip to points to the first symbol of w$
repeat forever begin
    let s be the state on top of the stack and a the symbol pointed to by ip;
     if action[sm, ai] = shift s' then begin
                push a then s' on top of the stack;
                advance ip to point to the next input symbol
    end
    else if action[sm, ai] = reduce A → β then begin
                pop 2*| β | symbols off the stack;
                let s' be the state now on top of the stack;
                push A then goto[s', A] on top of the stack;
                output the production A → β;
    end
    else if action[sm, ai] = accept then
                return
    else error()
end
```

LR Parsing Algorithm

- **Example-**

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

	Action Table							Goto Table		
state	**id**	**+**	*****	**(**	**)**	**$**		**E**	**T**	**F**
0	s5			s4				1	2	3
1		s6				acc				
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) $E \rightarrow E+T$ 2) $E \rightarrow T$

3) $T \rightarrow T*F$ 4) $T \rightarrow F$

5) $F \rightarrow (E)$ 6) $F \rightarrow id$

Stack	Input	Action	Output
(1) 0	id*id+id$	shift 5	
(2) 0id5	*id+id$	reduce by F→id	F→id
(3) 0F3	*id+id$	reduce by T→F	T→F
(4) 0T2	*id+id$	shift 7	
(5) 0T2*7	id+id$	shift 5	
(6) 0T2*7id5	+id$	reduce by F→id	F→id
(7) 0T2*7F10	+id$	reduce by T→T*F	T→T*F
(8) 0T2	+id$	reduce by E→T	E→T
(9) 0E1	+id$	shift 6	
(10) 0E1+6	id$	shift 5	
(11) 0E1+6id5	$	reduce by F→id	F→id
(12) 0E1+6F3	$	reduce by T→F	T→F
(13) 0E1+6T9	$	reduce by E→E+T	E→E+T
(14) 0E1	$	accept	

state	id	+	*	(	)	$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
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				

Think and Write

Parse the string id*id using LR parsing table

state	id	+	*	(	)	$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
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				

Think and Write

Stack	Input	Action	Output
0	id*id$	shift 5	
0id5	*id$	reduce by F→id	F→id
0F3	*id$	reduce by T→F	T→F
0T2	*id$	shift 7	
0T2*7	id$	shift 5	
0T2*7id5	$	reduce by F→id	F→id
0T2*7F10	$	reduce by T→T*F	T→T*F
0T2	$	reduce by E→T	E→T
0E1	$	accept	

1) $E \rightarrow E+T$ 2) $E \rightarrow T$

3) $T \rightarrow T*F$ 4) $T \rightarrow F$

5) $F \rightarrow (E)$ 6) $F \rightarrow id$

state	id	+	*	(	)	$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
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 Grammar

- A grammar for which an SLR parser can be constructed is said to be an **SLR grammar**.
- **LR(0) item** - An **LR(0) item** of a grammar G is a production of G with a dot at the some position of the right side.

  Example: $A \rightarrow XYZ$ Possible LR(0) Items: $A \rightarrow .XYZ$ (1)
  $A \rightarrow X.YZ$ (2)
  $A \rightarrow XY.Z$ (3)
  $A \rightarrow XYZ.$ (4)

- The production $A \rightarrow \varepsilon$ generates only one item $A \rightarrow .$
- A collection of sets of LR(0) items called as the **canonical LR(0) collection** is the basis for constructing SLR parsers.

SLR Grammar

- **Augmented Grammar** – If G is a grammar with start symbol S then G', the augmented grammar for G is with a new production rule S'→S where S' is the new starting symbol.

  Example – E → E + T | T

  T → T * F | F

  F → (E) | id

  So augmented grammar is

  E' → E

  E → E + T | T

  T → T * F | F

  F → (E) | id

SLR Grammar

- **The Closure Operation**

  If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from I by the two rules:

  1. Initially, every LR(0) item in I is added to closure(I).
  2. If $A \rightarrow \alpha.B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma$ will be in the closure(I). We will apply this rule until no more new LR(0) items can be added to closure(I).

SLR Grammar

- **The Closure Operation**

Example -

$E' \rightarrow E$
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

closure({E' → .E}) =

{ E' → .E ← kernel items
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id }

SLR Grammar

- **The Closure Operation**

```
function closure(I);
    begin
        J := I;
        repeat
                for each item A → α . B β in J and each production B → γ of G
                such that B → .γ is not in J do
                        add B → .γ is to J;
        until no more items can be added to J;
        return J
    end
```

- ✓ **Kernal Item –** include the initial item, S' → .S and all item whose dots are not at the left end.
- ✓ **Nonkernal Item** – have their dots at the left end.

SLR Grammar

- **The Goto Operation**
  - ✓ goto(I, X) is defined to be the closure of the set of all items [A → αX.β] such that [A → α.Xβ] is in I.

I = {E' → .E,
E → .E+T,
E → .T,
T → .T*F,
T → .F,
F → .(E),
F → .id }

goto(I,E) = { E' → E., E → E.+T }

goto(I,T) = { E → T., T → T.*F }

goto(I,F) = {T → F. }

goto(I,() = { F → (.E), E → .E+T, E → .T, T → .T*F, T → .F, F →.(E), F → .id }

goto(I,id) = { F → id. }

SLR Grammar

- **The Set-of-Items Construction**

```
procedure items(G');
    begin
        C := {closure({[S'→ .S]})};
        repeat
                for each set of item I in C and each ghrammar symbol X such that
                    goto(I, X) is not empty and not in C do
                        add goto(I, X) to C
        until no more sets of  items can be added to C;
    end
```

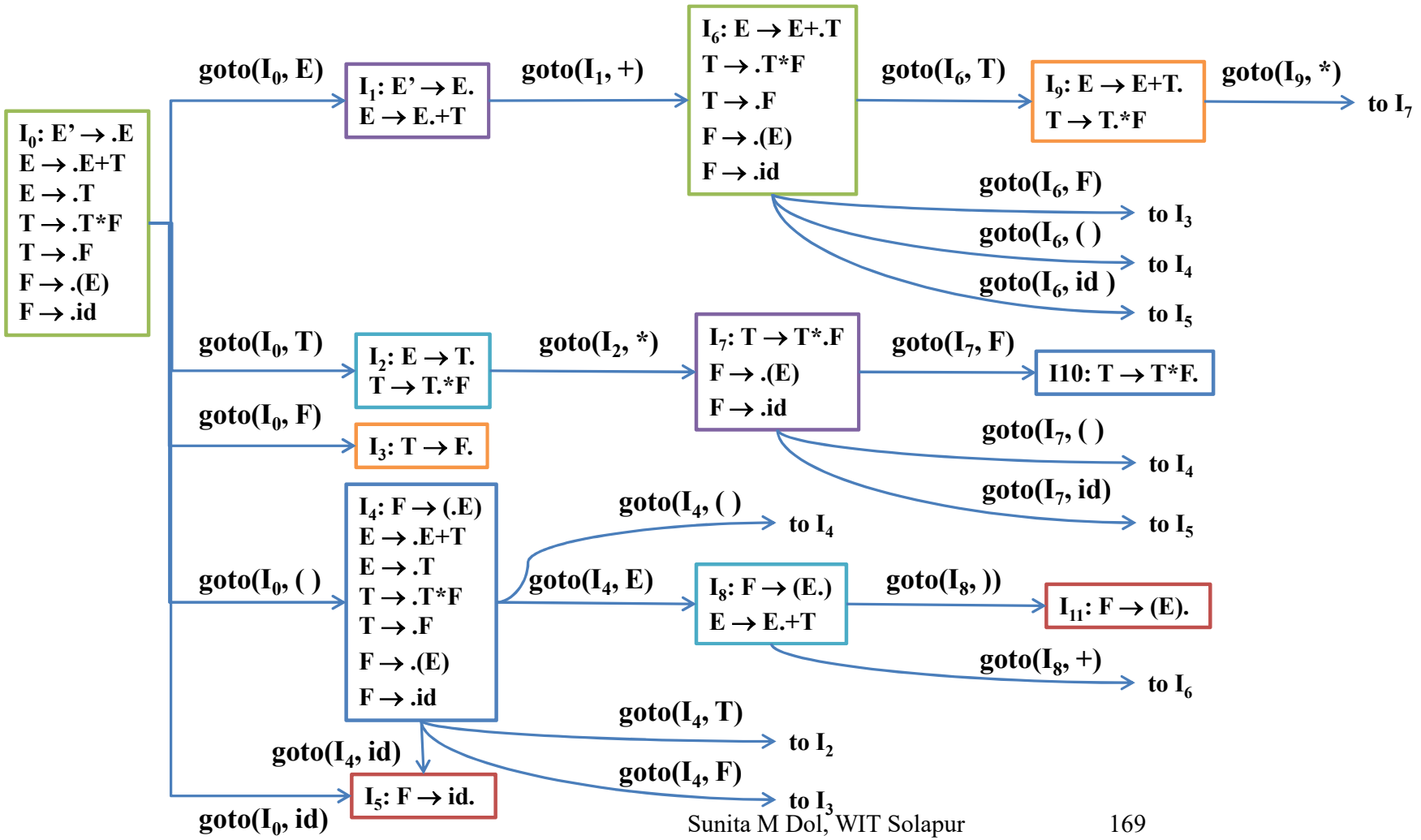
SLR Grammar

- **LR(0) item Example**

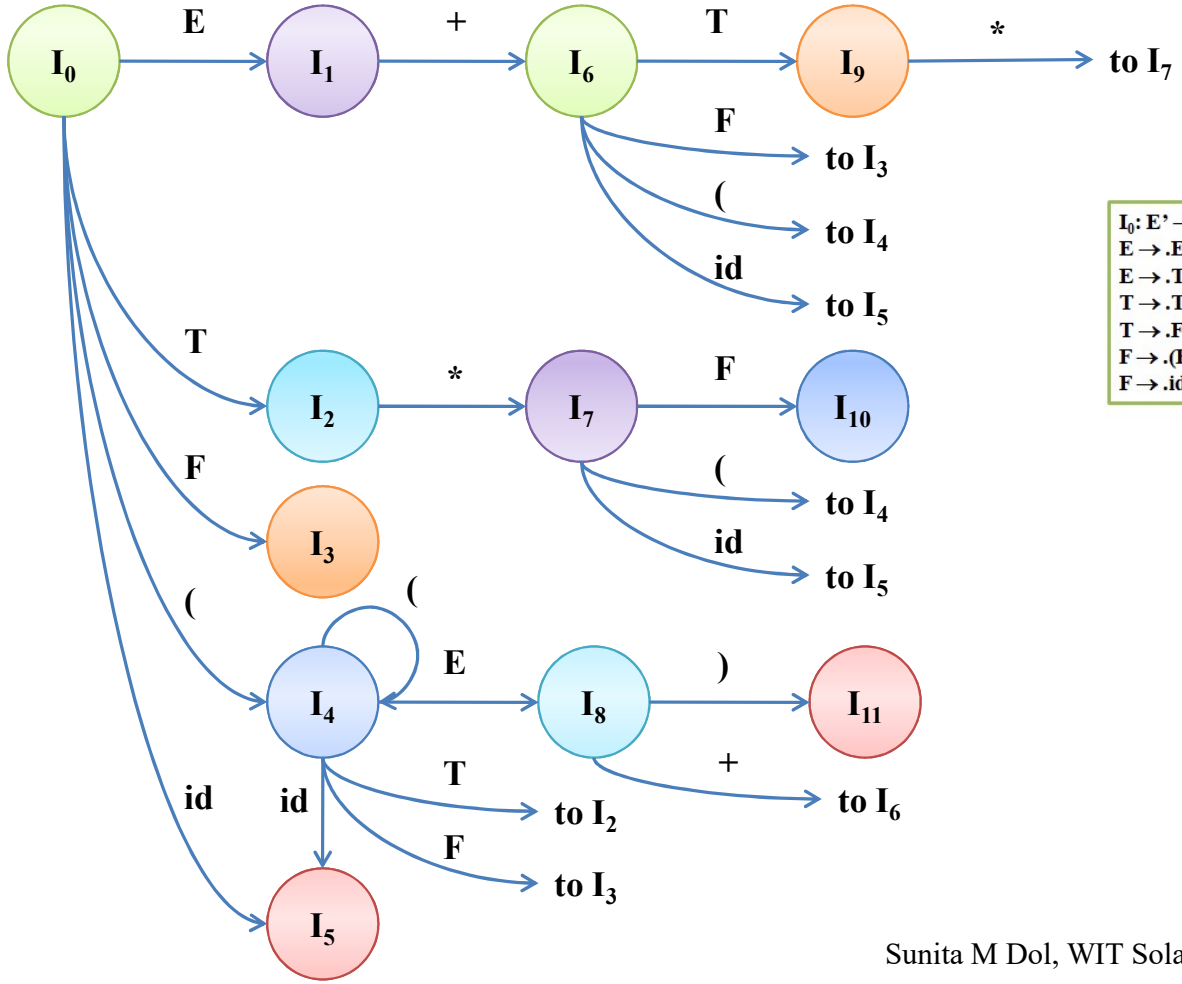
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

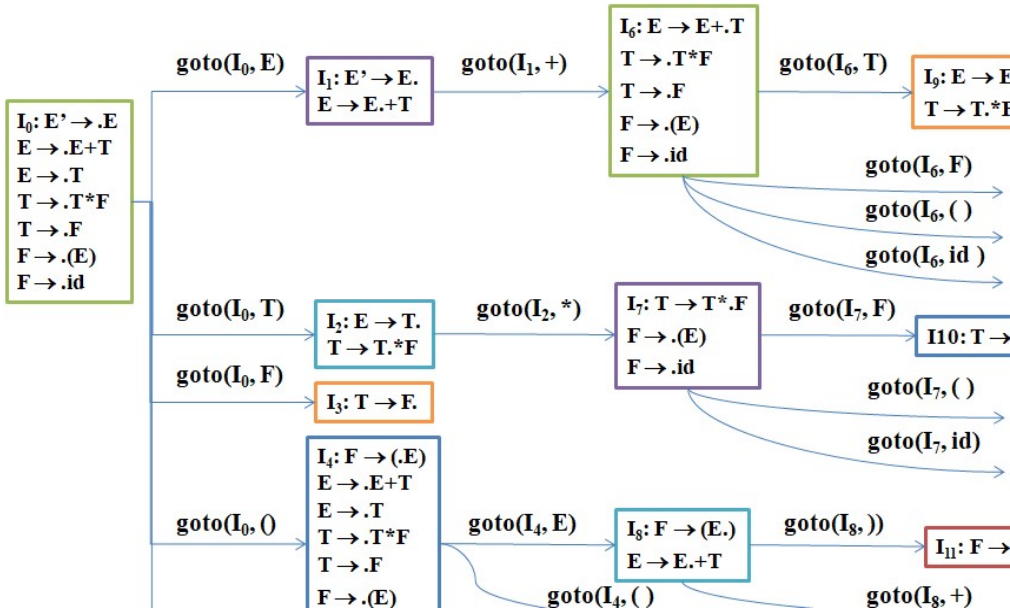
$E' \rightarrow E$
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

SLR Grammar


I0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id
goto(I0, E)
I1: E' → E.
E → E.+T
goto(I1, +)
I6: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id
goto(I6, T)
I9: E → E+T.
T → T.*F
goto(I9, *)
to I7
goto(I6, F)
to I3
goto(I6, ( )
to I4
goto(I6, id)
to I5
goto(I0, T)
I2: E → T.
T → T.*F
goto(I2, *)
I7: T → T*.F
F → .(E)
F → .id
goto(I7, F)
I10: T → T*F.
goto(I7, ( )
to I4
goto(I7, id)
to I5
goto(I0, F)
I3: T → F.
goto(I0, ( )
I4: F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id
goto(I4, ( )
to I4
goto(I4, E)
I8: F → (E.)
E → E.+T
goto(I8,))
I11: F → (E).
goto(I8, +)
to I6
goto(I4, T)
to I2
goto(I4, F)
to I3
goto(I4, id)
I5: F → id.
goto(I0, id)

Transition diagram of DFA D





Think and Write

Find LR(0) items for following grammar

$S \rightarrow L = R$

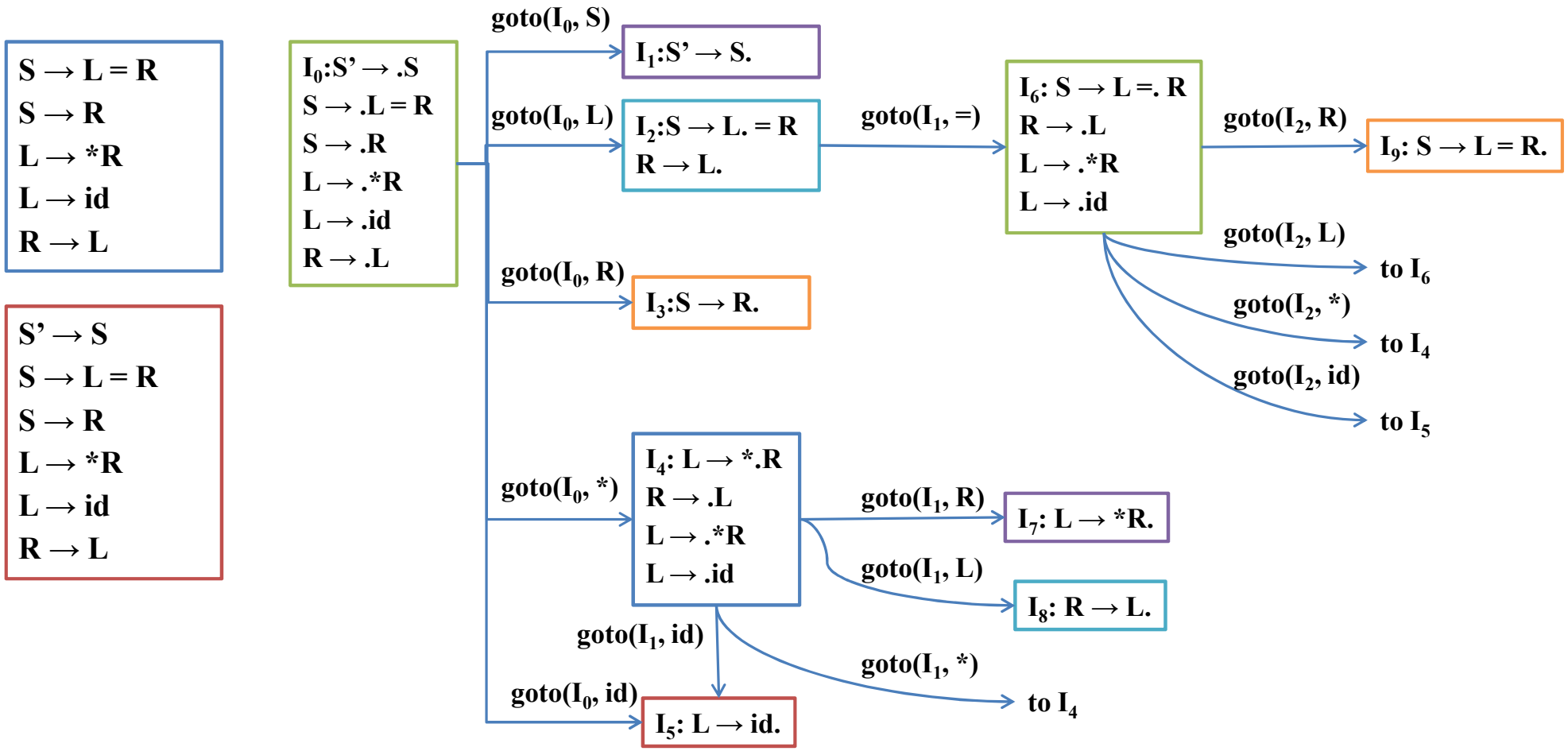
$S \rightarrow R$

$L \rightarrow *R$

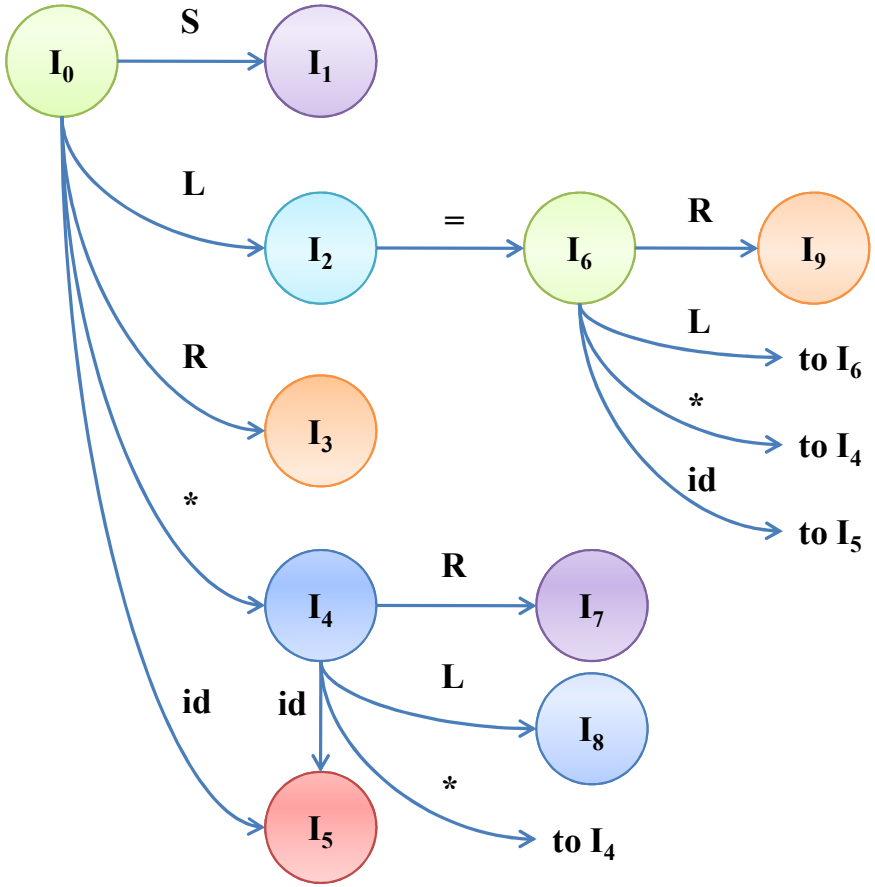
$L \rightarrow id$

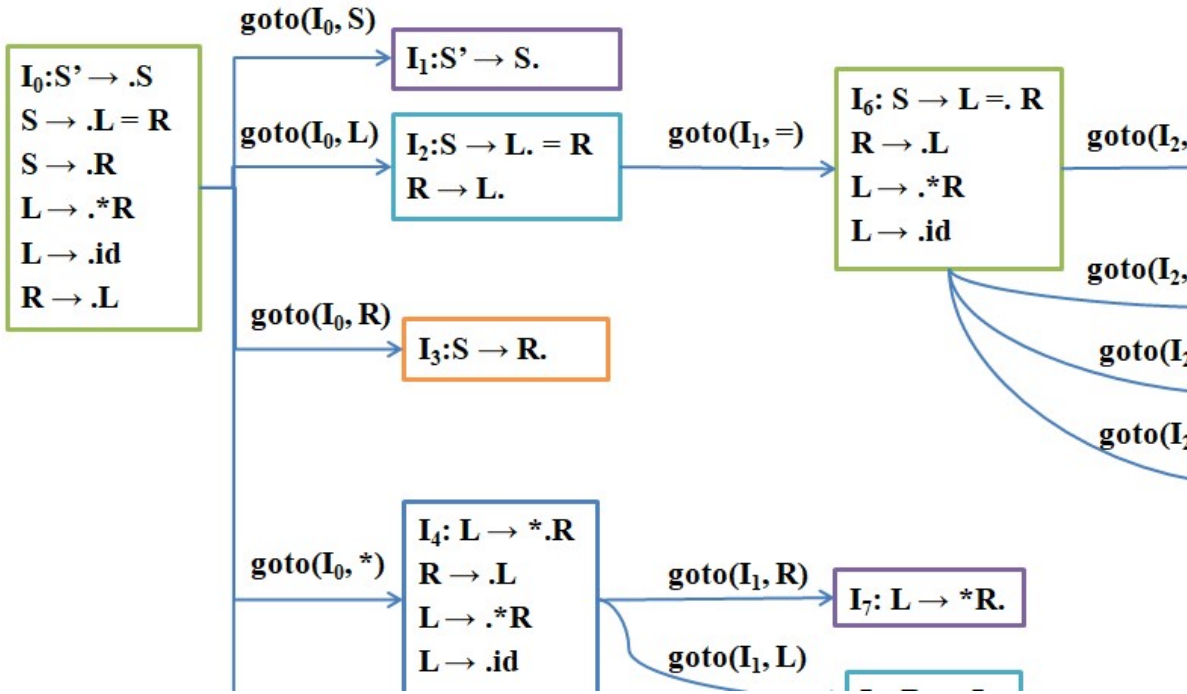
$R \rightarrow L$

Think and Write



Think and Write





Constructing SLR Parsing Table

1. Construct the canonical collection of sets of LR(0) items for G'.

   $C \leftarrow \{I_0,\ldots,I_n\}$

2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha.a\beta$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j.***
   - If $A \rightarrow \alpha.$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ for all a in FOLLOW(A) where $A \neq S'$.
   - If $S' \rightarrow S.$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).

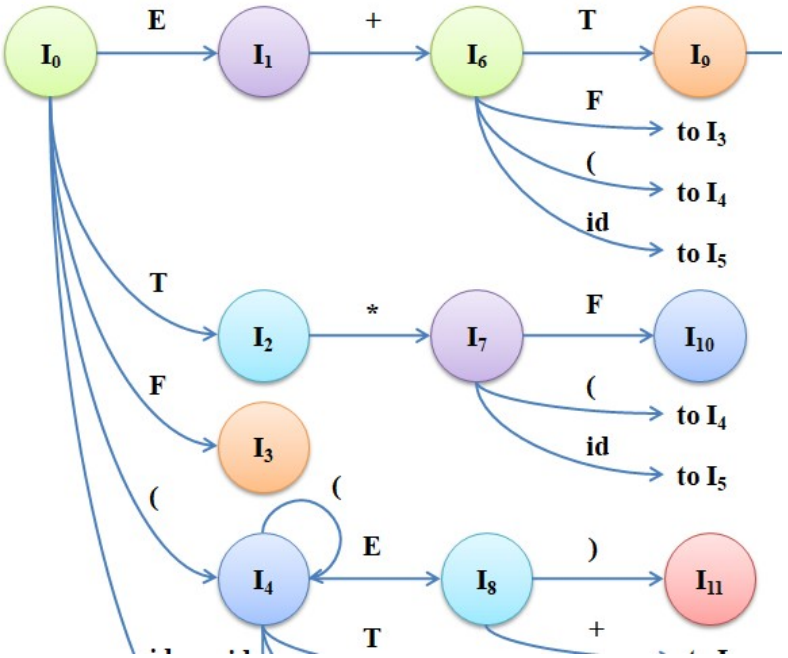
Constructing SLR Parsing Table

3. Create the parsing goto table
   - for all non-terminals A, if goto(I_i,A)=I_j then goto[i,A]=j

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains S'$\rightarrow$.S

Constructing SLR Parsing Table



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

	Action Table							Goto Table		
state	id	+	*	(	)	$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
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				

Constructing SLR Parsing Table

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Action Table | Goto

state	id	+	*	(	)	$	E
0	s5			s4			1
1		s6				acc	
2		r2	s7		r2	r2	
3		r4	r4		r4	r4	
4	s5			s4			8
5		r6	r6		r6	r6	
6	s5			s4			
7	s5			s4			
8		s6			s11		

References

- Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullmann “Compilers- Principles, Techniques and Tools”, Pearson Education.

Thank You !!!