COMP409: COMPILER DESIGN

3. SYNTAX ANALYSIS

Instructor: Sushil Nepal, Assistant Professor

Block: 9-308

Email: sushilnepal@ku.edu.np

Contact: 9851-151617

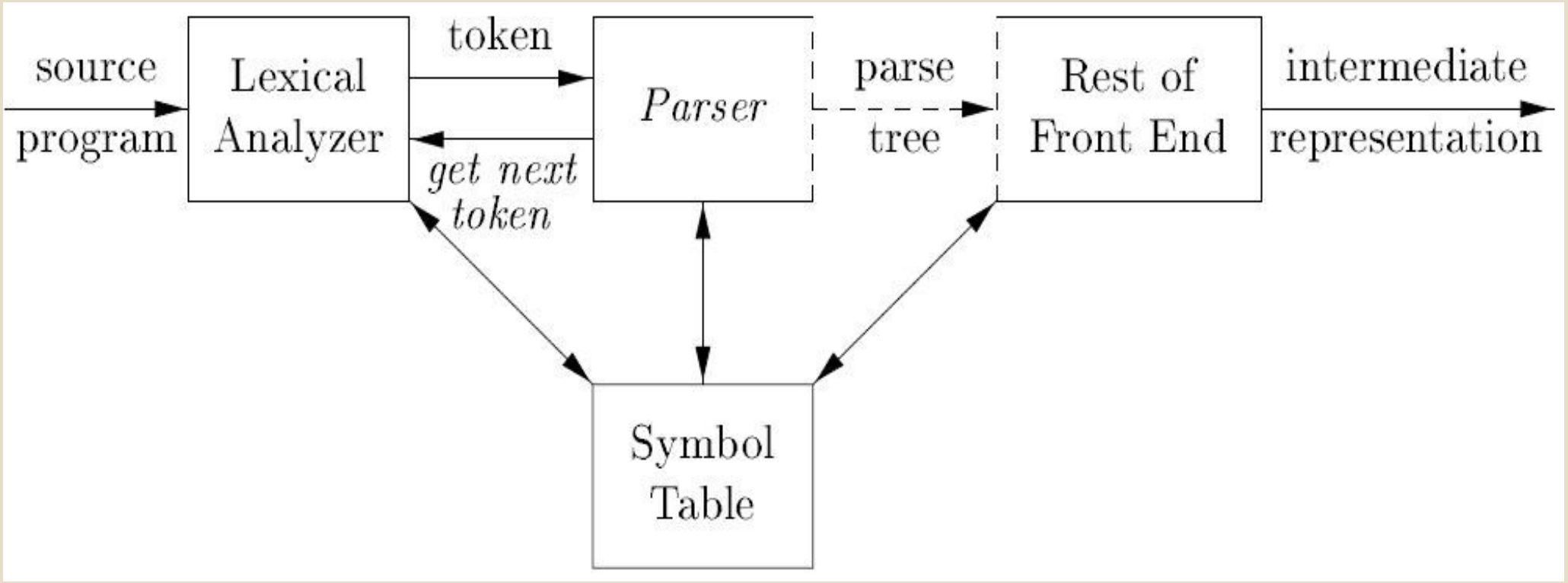
Introduction

- All programming languages have certain syntactic structures
- We need to verify that the source code written for a language is syntactically valid
- The validity of the syntax is checked by the syntax analysis
- Syntaxes are represented using context free grammar (CFG), or Backus Naur Form (BNF)
- Parsing is the act of performing syntax analysis to verify an input program's compliance with the source language

Introduction

- The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens
- A by-product of this process is typically a tree that represents the structure of the program
- Parsing is the act of checking whether a grammar “accepts” an input text as valid (according to the grammar rules)
- It determines the exact correspondence between the text and the rules of given grammar

The Role of the Parser



The Role of the Parser

- Analyzes the context free syntax
- Generates the parse tree
- Determines errors and tries to handle them

Types of Parser

- There are three general types of parsers for grammars: universal, top-down and bottom-up
- **Universal Parser**
  - Can parse any kind of grammars
  - Too inefficient to use in production compilers
  - E.g. of universal parsing methods: CYK algorithm, Earley's algorithm
- The methods commonly used in compilers can be classified as being either top-down or bottom-up

Types of Parser

- **Top-Down Parser**
  - the parse tree is created top to bottom, starting from the root
  - LL for top-down parsing
- **Bottom-Up Parser**
  - the parse is created bottom to top; starting from the leaves
  - LR for bottom-up parsing

Types of Parser

- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time)
- Efficient top-down and bottom-up parsers can only be implemented for sub-classes of context-free grammars

Error Handling

- Every phase of the compiler is prone to errors
- Syntax analysis and semantic analysis are the phases that are the most common sources of errors
- Syntactic errors include misplaced semicolons or extra or missing braces; that is, “{”or “}”
- As another example, in C or Java, the appearance of a **case** statement without an enclosing **switch** is a syntactic error
- If error occurs, the process should not terminate but instead report the error and try to advance

Error Recovery Techniques

- **Panic Mode Recovery**
  - With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found, say a semicolon
  - May skip errors if there are more than one error in the sentence
  - It has the advantage of simplicity, and, unlike some methods, is guaranteed not to go into an infinite loop

Error Recovery Techniques

- **Phrase-Level Recovery**
  - On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue
  - A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon

Error Recovery Techniques

- **Error Productions**
  - By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs
  - A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing
  - The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input

Error Recovery Techniques

- **Global Correction**
  - We would like a compiler to make as few changes as possible in processing an incorrect input string
  - There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction
  - Given an incorrect input string **X** and grammar **G**, these algorithms will find a parse tree for a related string **Y**, such that the number of insertions, deletions, and changes of tokens required to transform **X** into **Y** is as small as possible

Error Recovery Techniques

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

Context Free Grammar

- Most of the programming languages have recursive structures that can be defined by Context Free Grammar (CFG)
- CFG can be defined as 4 – tuple (V, T, P, S), where

  V → finite set of Variables or Non-terminals that are used to define the grammar denoting the combination of terminal or non-terminals or both

  T → Terminals, the basic symbols of the sentences; they are indivisible units

  P → Production rule that defines the combination of terminals or non-terminals or both for particular non-terminal

  S → It is the special non-terminal symbol called start symbol

- Example: Grammar to define a palindrome string over binary string S

  → 0S0 | 1S1

  S → 0 | 1

- Example: Grammar to define an infix expression E

  → E A E | (E) | -E | id

  A → + | - | * | / | ^

E and A are non terminals and E as start symbol. Remaining are non terminals.

Notational Conventions

- Lowercase letters, symbols (like operators), bold string are used for denoting terminals. Eg :- a, b, c, 0, 1 $\in$ T
- Uppercase letters, italicized strings are used for denoting non-terminals. Eg :- A, S, B, C $\in$ V
- Lowercase Greek letters (α, β, γ, δ) are used to denote the terminal, non-terminal or combination of both
- Production of the form A $\rightarrow$ a | b, is read as "A produces a or b"

Derivations

- Verification of a sentence defined by the grammar is done using production rules to obtain the particular sentence by expansion of non-terminals
- This process is known as a **derivation**
- $E \Rightarrow E+E$ means $E+E$ derives from E
  - we can replace E by $E+E$
  - to able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar
- $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$
- A sequence of replacements of non-terminal symbols is called a **derivation** of id+id from E

Derivations

- The term $\alpha \Rightarrow \beta$ is used to denote that β can be **derived** from α
- Similarly, $\alpha \Rightarrow^* \beta$ denotes derivation using zero or more rules and $\alpha \Rightarrow^+ \beta$ denotes derivation using one or more rules
- L(G) is the **language** of G (the language generated by G) which is a set of sentences
- A **sentence** of L(G) is a string of terminal symbols of G

Derivations

- If S is a start symbol of G then, *w* is a sentence of L(G) if and only if S $\Rightarrow^{*}$ *w*, where *w* is a string of terminals of G
- If G is a context free grammar, then L(G) is a context free language
- Two grammars are equivalent if they produce the same language
- For S $\Rightarrow$ α
  - If α is a combination of terminals and non-terminals, it is called as a **sentential** form of G
  - If α contains terminals only, it is called as a **sentence** of G

Derivations

- If we always choose the left-most non-terminal in each derivation step, this derivation is called **left-most derivation**
- If we always choose the right-most non-terminal in each derivation step, this derivation is called **right-most derivation**

Example:

Consider a grammar G (V, T, P, S), where

V → {E}

T → {+, *, -, (,), id}

P → {E → E + E, E → E * E, E → (E), E → -E, E → id } S → {E}

For *id * id + id*,

E ⇒ E * E ⇒ id * E ⇒ id * E + E ⇒ id * id + E ⇒ id * id + id

This is left-most derivation

Parse Tree

- The pictorial representation of the derivation can be depicted using the parse tree
- In parse tree internal nodes represent non-terminals and the leaves represent terminals
- Consider the grammar above:

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

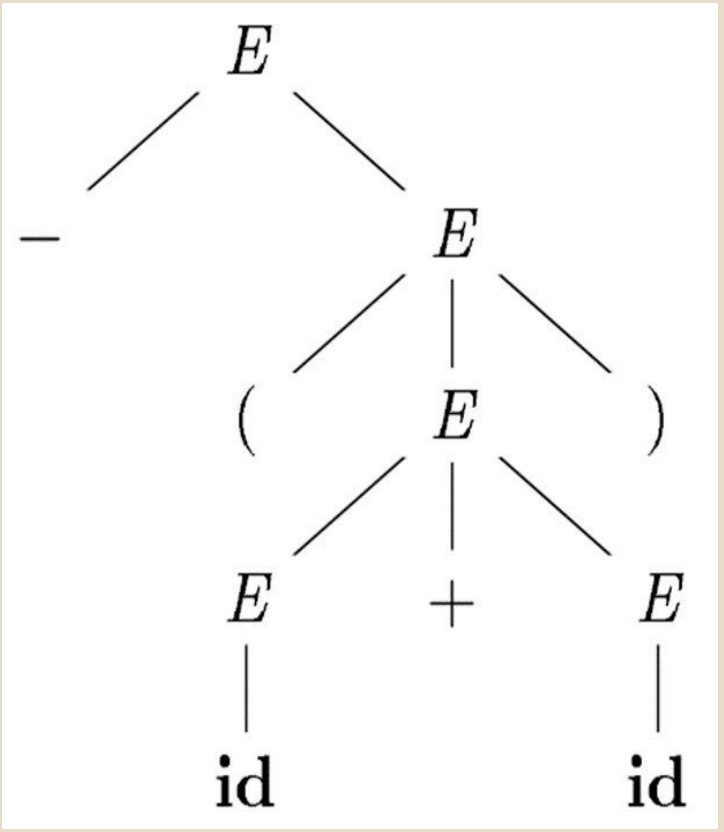
- The string -(id + id) is a sentence of this grammar because there is a derivation:

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$

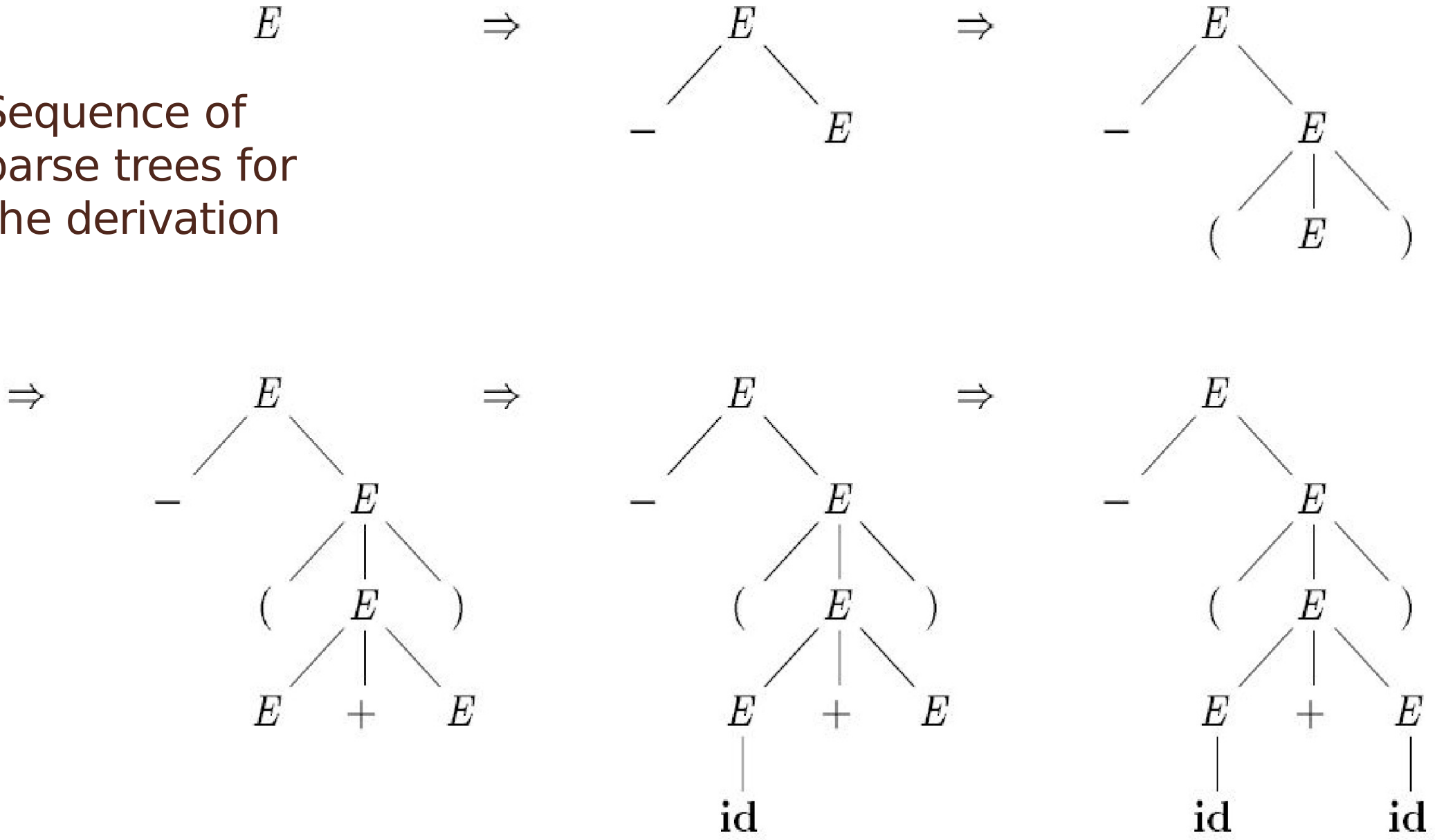
Parse Tree

Parse tree for:

-(id + id)



Sequence of parse trees for the derivation



Ambiguity

- A grammar is said to be **ambiguous** if it can produce a sentence in more than one way
- If there is more than one parse tree for a sentence or derivation (left or right) with respect to the given grammar, then the grammar is said to be **ambiguous**

Ambiguity

- Consider the grammar above:

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

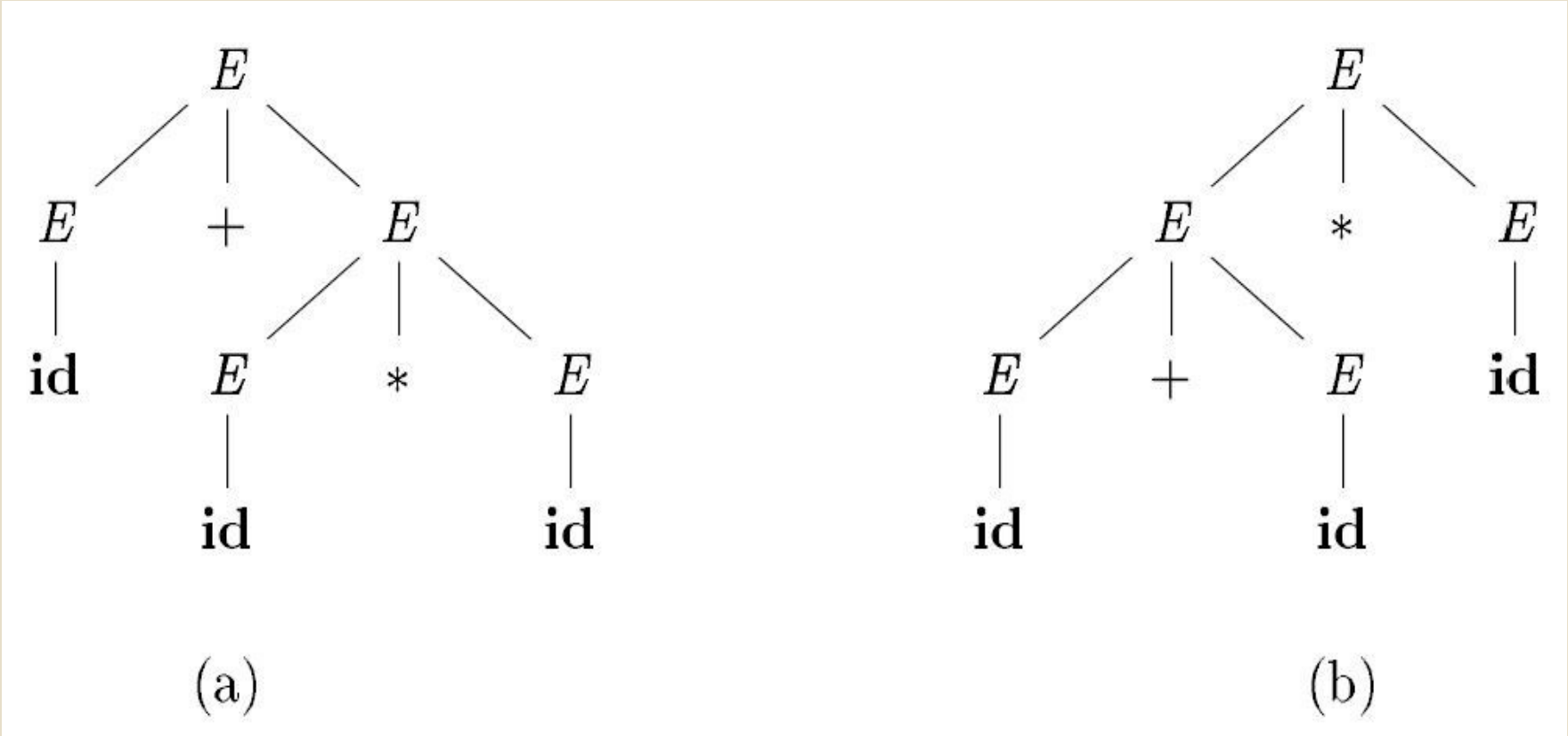
- Consider the string: **id + id * id**
- It has two distinct derivations:

(a) $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$

(b) $E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$

- So, this grammar is ambiguous

Ambiguity



Left Recursion

- The grammar with recursive non-terminal at the left of the production is called left recursive grammar
- A grammar is left recursive if it has a non – terminal “A” such that there is a derivation, $A \Rightarrow^{+} A\alpha$ for some string α

  - Eg: $E \rightarrow E + T \mid T$

    $T \rightarrow T * F \mid F$

    $F \rightarrow id \mid (E)$

Left Recursion

- Left recursion causes recursive descent parser to go into infinite loop
- Top down parsing techniques cannot handle left recursive grammars
- So, we have to convert our left recursive grammar into one which is not left recursive
- The left recursion may appear in a single derivation(called immediate left recursion), or may appear in more than one step of the derivation

Left Recursion

- Example of non immediate left recursion:

$S \rightarrow Aab \mid c$

$A \rightarrow Sc \mid c$

- This grammar is not immediately left recursive, but it is still left recursive
- For instance, $S \Rightarrow Aab \Rightarrow Scab$ causes left recursion

Removing Left Recursion

- If we have a grammar of the form $A \rightarrow A\alpha \mid \beta$, the rule for removing left recursion is

$$A \rightarrow \beta A'$$

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

- This is the equivalent non-recursive grammar

Removing Left Recursion

- Any immediate left-recursion can be eliminated by generalizing the above
- For a group of productions of the form:

$$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_n | \beta_1 | \beta_2 | \ldots | \beta_m$$

we replace the A-production by

$$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_m A'$$

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_n A' | \varepsilon$$

Removing Left Recursion

Algorithm:

(Note that the resulting non-left-recursive grammar may have ε–productions)

Input → Grammar G

Output → Equivalent grammar with no left recursion

Arrange non terminals in some order $A_1, A_2, \ldots\ldots, A_n$

for i=1 to n do

  for j=1 to i-1 do

    replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \alpha_1 \gamma$ $|\ldots\ldots|\ \alpha_k \gamma$,

    where $A_j \rightarrow \alpha_1 |\ldots\ldots| \alpha_k$ are all current A_j-productions

  end do

  eliminate the immediate left recursions among A_i-productions

end do

Example

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

We order the non terminals S, A. There is no immediate left recursion among the S-productions, so nothing happens during the outer loop for i = 1.

For i = 2, we substitute for S in $A \rightarrow Sd$

Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$

Then we have, $A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

Now, remove the immediate left recursions

$A \rightarrow bdA' \mid \varepsilon A'$

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

So, the resulting equivalent grammar with no left recursion is

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

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

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

Left Factoring

- 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
- In general, if $A \rightarrow \alpha\beta_1 | \alpha\beta_2$ are two A-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$

Left Factoring

- However, we may defer the decision by expanding A to A'
- Then, after seeing the input derived from α, we expand A' to β_1 or β_2
- That is, left-factored, the original productions become $A \rightarrow \alpha A'$
- $A' \rightarrow \beta_1 | \beta_2$

- Algorithm: Left factoring a grammar
- INPUT: Grammar G
- OUTPUT: An equivalent left-factored grammar
- METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives.

If $\alpha \neq \varepsilon$, replace all of the A-productions $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots . | \alpha\beta_n | \gamma$, where γ represents all alternatives that do not begin with α, by

$$A \rightarrow \alpha A' | \gamma$$

$$A' \rightarrow \beta_1 | \beta_2 | \ldots . | \beta_n$$

- Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

- <u>Example:</u> The “dangling-else” problem:

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

$$E \rightarrow b$$

- Here, **i, t,** and **e** stand for **if, then,** and **else**; E and S stand for “conditional expression” and “statement”
- Left-factored, this grammar becomes:

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

$$S' \rightarrow e\ S \mid \varepsilon$$

$$E \rightarrow b$$

- Thus, we may expand S to iEtSS′ on input i, and wait until iEtS has been seen to decide whether to expand S′ to eS or to ε

Parsing

- Given a stream of input tokens, **parsing** involves the process of "reducing" them to a non-terminal
- The input string is said to represent the non-terminal it was reduced to
- Parsing can be either **top-down** or **bottom-up**
- **Top-down** parsing involves generating the string starting from the first non-terminal and repeatedly applying production rules.
- **Bottom-up** parsing involves repeatedly rewriting the input string until it ends up in the first non-terminal of the grammar.

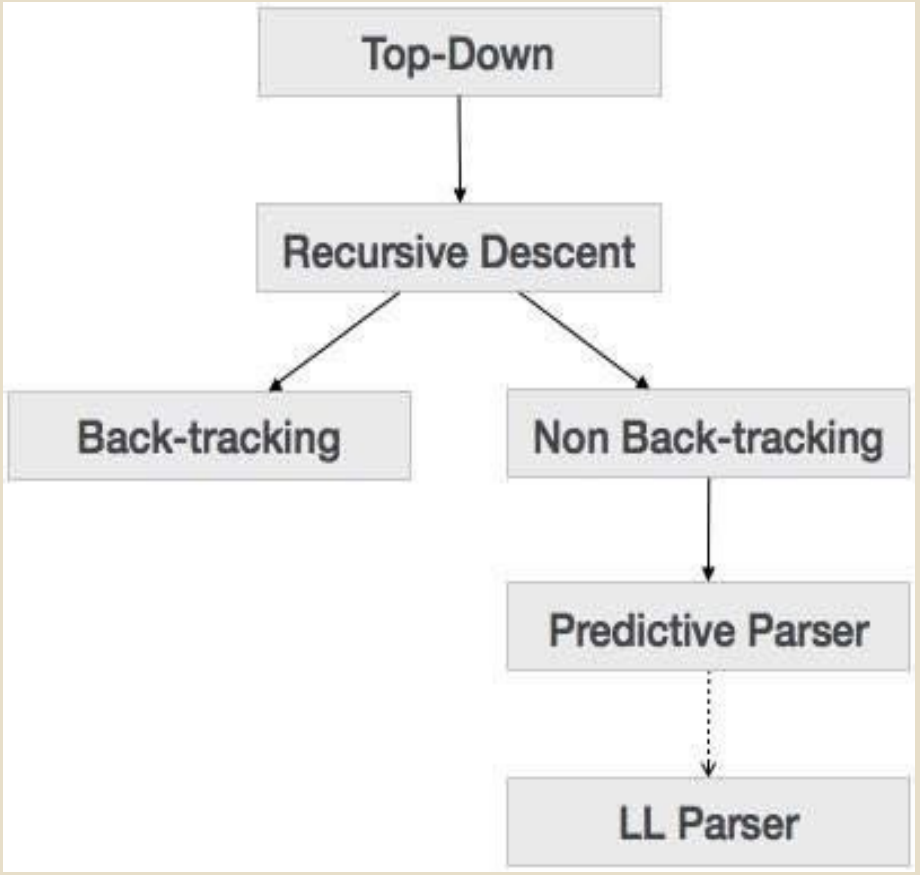
Top-Down Parsing

- 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
- Top-down parsing can be viewed as finding a leftmost derivation for an input string
- Starting at the start non-terminal symbol, the use of production rules leads to the input string

Top-Down Parsing

- At each step of a top-down parse, the key problem is that of determining the production to be applied for a nonterminal, say A
- Once an A-production is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string

Top-Down Parsing



Recursive-Descent Parsing

- A recursive-descent parsing program consists of a set of procedures, one for each nonterminal
- Recursive-descent parser may require backtracking to find the correct A-production to be applied
- Start with the starting non-terminal and use the first production, verify with input and if there is no match, backtrack and apply another rule
- Backtracking is rarely needed to parse programming language constructs, so backtracking parsers are not seen frequently

Recursive-Descent Parsing

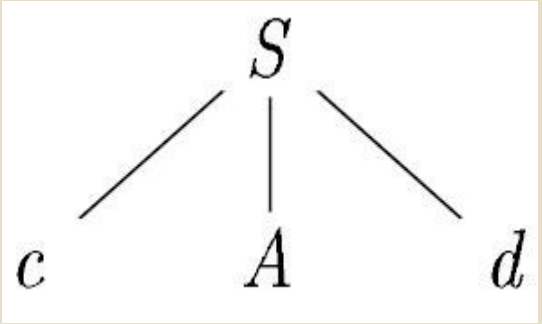
- Consider the grammar

  $S \rightarrow cAd$

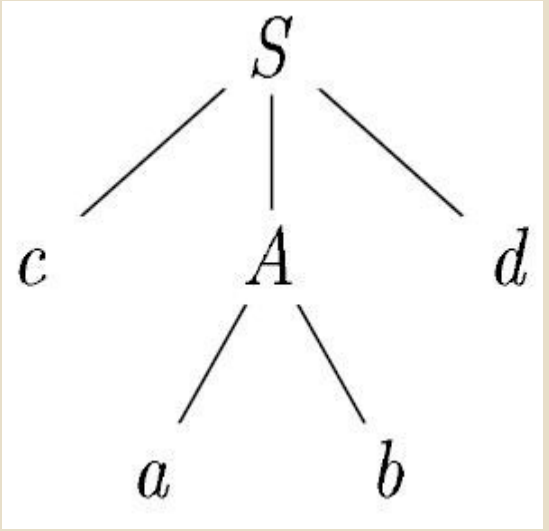
  $A \rightarrow ab|a$

- To construct a parse tree top-down for the input string w = cad, begin with a tree consisting of a single node labeled S, and the input pointer pointing to **c**, the first symbol of w

- S has only one production, so we use it to expand S and obtain the following tree:

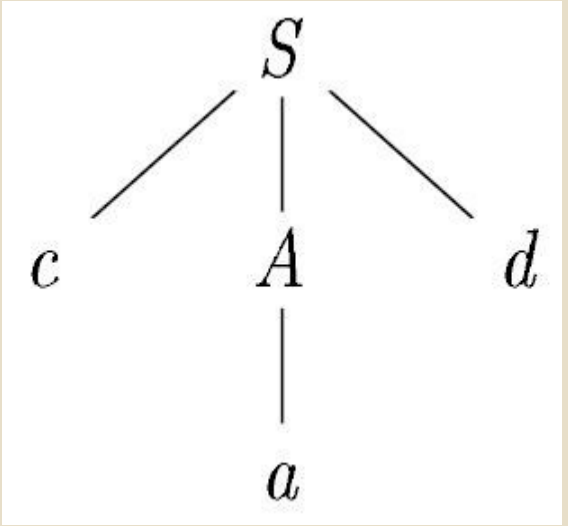


- The leftmost leaf, labeled **c**, matches the first symbol of input w, so we advance the input pointer to **a**, the second symbol of w, and consider the next leaf, labeled A
- Now, we expand A using the first alternative A → ab to obtain the subsequent tree



- We have a match for the second input symbol, **a**, so we advance the input pointer to **d**, the third input symbol, and compare **d** against the next leaf, labeled **b**
- Since **b** does not match **d**, we report failure and go back to A to see whether there is another alternative for A that has not been tried, but that might produce a match

- In going back to A, we must reset the input pointer to position 2, the position it had when we first came to A
- The second alternative for A produces the following tree:



- The leaf '**a**' matches the second symbol of w and the leaf **d** matches the third symbol
- Since we have produced a parse tree for w, we halt and announce successful completion of parsing

Recursive-Descent Parsing

- A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop
- That is, when we try to expand a nonterminal A, we may eventually find ourselves again trying to expand A without having consumed any input

Algorithm:

1.Use two pointers: **iptr** for pointing the input symbol to be read and **optr** for pointing the symbol of output string, initially start symbol S.

2.If the symbol pointed by **optr** is a non-terminal, use the first unexpanded production rule for expansion.

3.While the symbol pointed by **iptr** and **optr** is same increment the both pointers.

4. The loop in the above step terminates when

a. there is a non-terminal at the output (case A)

b. it is the end of input (case B)

c. unmatching terminal symbol pointed by **iptr** and **optr** is seen (case C)

5. If (A) is true, goto step 2

6. If (B) is true, terminate with success

7.If (C) is true, decrement both pointers to the place of last non-terminal expansion(backtrack) and goto step 2

8.If there is no more unexpanded production left and (B) is not true, report error

Example:

Consider a grammar:

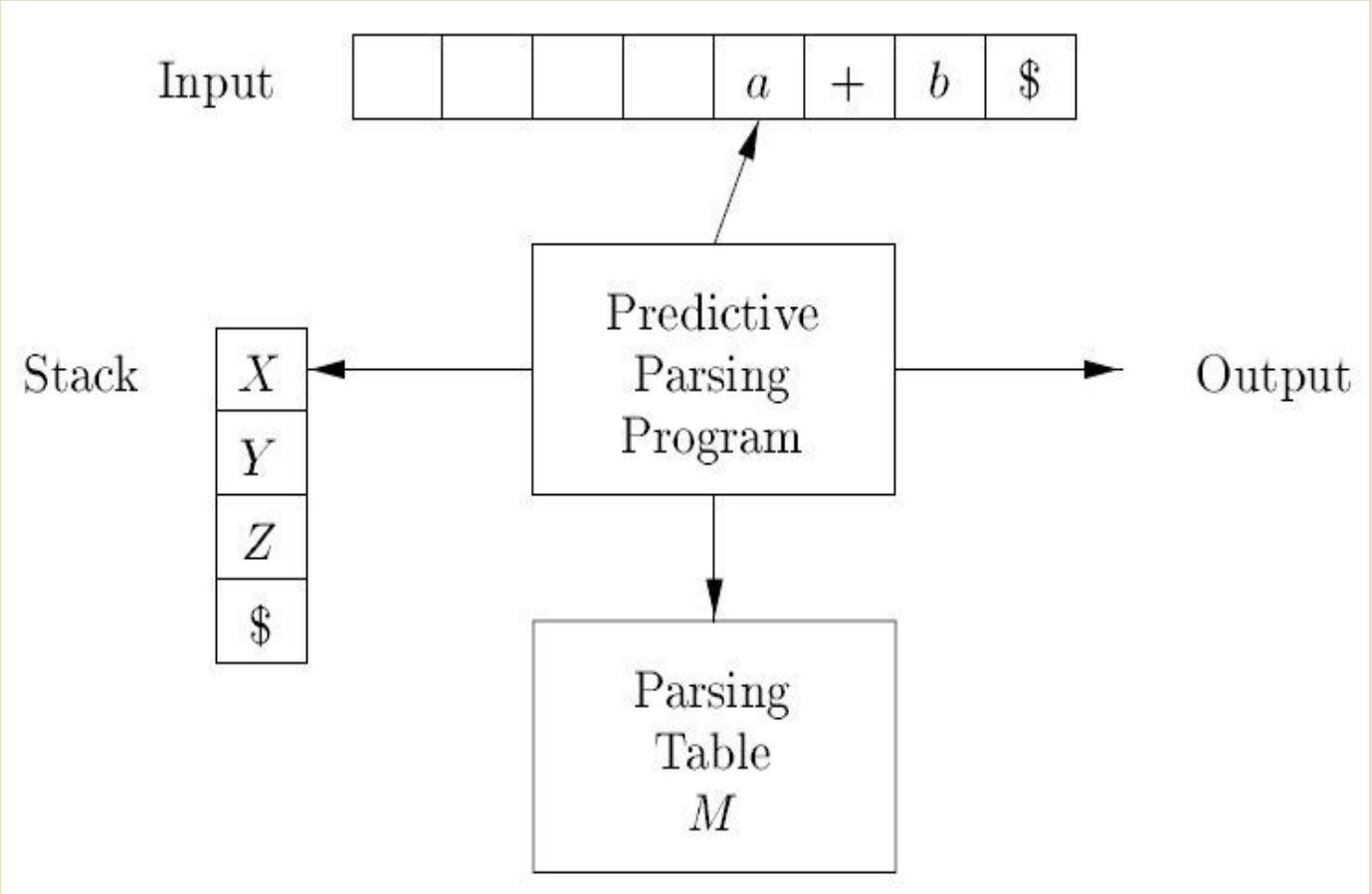
S → cAd

A → ab|a

Input string is “cad”

Input	Output	Rules Fired
(iptr)cad	(optr)S	[Rule 2, Try S → cAd]
(iptr)cad	(optr)cAd	[Rule 3, Match c]
c(iptr)ad	c(optr)Ad	[Rule 5, Try A → ab]
c(iptr)ad	c(optr)abd	[Rule 3, Match a]
ca(iptr)d	ca(optr)bd	[Rule 7, dead end, backtrack]
c(iptr)ad	c(optr)Ad	[Rule 5, Try A → a]
c(iptr)ad	c(optr)ad	[Rule 3, Match a]
ca(iptr)d	ca(optr)d	[Rule 3, Match d]
cad(iptr)	cad(optr)	[Rule 6, Terminate with success]

Nonrecursive Predictive Parsing



FIRST and FOLLOW

- The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G

- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol

- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols
  - if α ⇒* ε i.e., α derives to ε, then ε is also in FIRST(α)

- **FOLLOW(A)** is the set of the terminals which occur immediately after(follow) the non-terminal A in the strings derived from the starting symbol
  - a terminal **a** is in FOLLOW(A) if S ⇒* αAaβ
  - $ is in FOLLOW(A) if S ⇒ αA (i.e., if A can be the last symbol in a sentential form)

Computing FIRST [FIRST(α) = {the set of terminals that begin all strings derived from α}]

1. If X is ε then FIRST(X)={ε}
2. If X is a terminal symbol then FIRST(X) = {X}
3. If X is a non-terminal symbol and $X \rightarrow \varepsilon$ is a production rule then FIRST(X)=FIRST(X) ∪ ε
4. If X is a non-terminal symbol and $X \rightarrow Y_1 Y_2 \ldots Y_n$ is a production rule then

a) if a terminal **a** is in FIRST(Y_1) then FIRST(X) = FIRST(X) ∪ **a**

b) if a terminal **a** is in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then FIRST(X) = FIRST(X) ∪ **a**

Now, we can compute FIRST for any string $X_1X_2…X_n$ as follows:

Add to FIRST($X_1X_2…X_n$) all non-ε symbols of FIRST(X_1)

Also add the non-ε symbols of FIRST(X_2), if ε is in FIRST(X_1); the non-ε symbols of FIRST(X_3), if ε is in FIRST(X_1) and FIRST(X_2); and so on

Finally, add ε to FIRST($X_1X_2…X_n$) if, for all i, ε is in FIRST(X_i)

Computing FOLLOW [FOLLOW(A) = {the set of terminals that can immediately follow non-terminal A}]

1.Place $ in FOLLOW(S), where S is the start symbol, and $ is the input right endmarker

2.For every production B → αAβ, where α and β are any strings of grammar symbols and A is non-terminal, then everything in FIRST(β) except ε is placed on FOLLOW(A)

3.For every production B → αA, or a production B → αAβ, where FIRST(β) contains ε (i.e. β is nullable), then everything in FOLLOW(B) is added to FOLLOW(A)

E → TE′

E′ → +TE′ | ε

T → FT′

T′ → *FT′ | ε

F → (E) | id

FIRST(F) = {(, id}

FIRST(T′) = {* , ε}

FIRST(T) = {(, id}

FIRST(E′) = {+, ε}

FIRST(E) = {(, id}

FIRST(TE′) = {(, id}

FIRST(+TE′) = {+}

FIRST(FT′) = {(, id}

FIRST(ε) = {ε}

FIRST(*FT′) = {*}

Example 1:

FIRST((E)) = { (}

FIRST(id) = { id }

FOLLOW(E) = { $,) }

FOLLOW(E′) = { $,) }

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

FOLLOW(T′) = { +,) , $ }

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

Example 2:

S → AB

A → Ca | ε

B → BaAC | c

C → b | ε

Notice we have a left-recursive production that must be fix if we are to use LL(1) parsing:

B → BaAC | c

becomes

B → cB′

B′ → aACB′ | ε

Now the new grammar(G) is:

$S \rightarrow AB$

$A \rightarrow Ca \mid \varepsilon$

$B \rightarrow cB'$

$B' \rightarrow aACB' \mid \varepsilon$

$C \rightarrow b \mid \varepsilon$

It helps to first compute the nullable set (i.e., those non-terminals X that $X \Rightarrow^* \varepsilon$), since you need to refer to the nullable status of various nonterminals when computing the first and follow sets:

nullable(G) = {A, B′, C}

Predictive Parsing

- A Recursive Descent parser always chooses the first available production whenever encountered by a non-terminal
- This is inefficient and causes a lot of backtracking
- It also suffers from the left-recursion problem
- The recursive descent parser can work efficiently if there is no need of backtracking
- A predictive parser tries to predict which production produces the least chances of a backtracking and infinite looping

Predictive Parsing

- A predictive parser is characterized by its ability to choose the production to apply solely on the basis of the next input symbol and the current non-terminal being processed
- To enable this, the grammar must take a particular form called a LL(1) grammar
- The first "L" means we scan the input from left to right; the second "L" means we create a leftmost derivation; and the 1 means one input symbol of look ahead
- LL(1) has no left recursive productions and is left-factored

LL(1) Grammars

- Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1)
- The predictive top-down techniques(either recursive-descent or table-driven) require a grammar that is LL(1)
- One fully general way to determine if a grammar is LL(1) is to build the table and see if you have conflicts
- A grammar whose parsing table has no multiple defined entries is said to be LL(1) grammar

Constructing LL(1) Parsing Table

Input: LL(1) Grammar G

Output: Parsing Table M

for each production rule A → α of a grammar G

    for each terminal **a** in FIRST(α)

        add A → α to [A,**a**]

    if ε in FIRST(α)

    then

        for each terminal **a** in FOLLOW(A)

            add A → α to M[A,**a**]

    if ε in FIRST(α) and $ in FOLLOW(A) then

        add A → α to M[A,$]

make all undefined entries of the parsing table M as error

Example: Consider the grammar

G:

E → TE'

E' → +TE' | ε

T → FT'

T′ → *FT′| ε

F → (E) | id

FIRST(F) ={(, id}

FIRST(T′) ={*, ε }

FIRST(T) ={(, id}

FIRST(E′) ={+, ε }

FIRST(E) ={(, id}

FIRST(TE′) = {(, id}

FIRST(+TE′) = {+}

FIRST(ε) = { ε }

FIRST(FT′) = {(, id}

FIRST(*FT′) = {*}

FIRST(ε) = { ε }

FIRST((E)) = {(}

FIRST(id) = {id}

FOLLOW(E) ={$,)}

FOLLOW(E′) ={$,)}

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

FOLLOW(T′) ={+, $,)}

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

Now do for every production

$E \rightarrow TE'$	FIRST(TE') = {(, id}	$E \rightarrow TE'$ into M[E, (] and M[E, id]
$E' \rightarrow +TE'$	FIRST(+TE') = {+}	$E' \rightarrow +TE'$ into M[E', +]
$E' \rightarrow \varepsilon$	FIRST(ε) = { ε } but since ε in FIRST(ε) and FOLLOW(E') = {$,)}	$E' \rightarrow \varepsilon$ into M[E', $] and M[E',)]
$T \rightarrow FT'$	FIRST(FT') = {(, id}	$T \rightarrow FT'$ into M[T, (] and M[T, id]
$T' \rightarrow *FT'$	FIRST(*FT') = {*}	$T' \rightarrow *FT'$ into M[T', *]
$T' \rightarrow \varepsilon$	FIRST(ε) = { ε } but since ε in FIRST(ε) and FOLLOW(T') = {$,), +}	$T' \rightarrow \varepsilon$ into M[T', $], M[T',)] and M[T', +]
$F \rightarrow (E)$	FIRST((E)) = {(}	$F \rightarrow (E)$ into M[F, (]
$F \rightarrow id$	FIRST(id) = {id}	$F \rightarrow id$ into M[F, id]

- Final Parsing Table:

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

Nonrecursive Predictive Parsing

- A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls
- Non recursive predictive parsing is a table driven parser
- The table driven predictive parser has stack, input buffer, parsing table and output stream
- The input buffer contains the sentence to be parsed followed by $ as an end marker
- The stack contains symbols of the grammar

Nonrecursive Predictive Parsing

- Initially stack contains start symbol on top of $
- When the stack is emptied (i.e. only $ is left in the stack), parsing is completed
- Parsing table is two dimensional array M[A,a] containing information about the production to be used upon seeing the terminal at input and nonterminal at the stack
- Each entry in the parsing table holds the production rule
- Each column holds the terminal or $, and each row holds non terminal symbols

Nonrecursive Predictive Parsing

- Algorithm:
- INPUT: A string **w** and a parsing table **M** for grammar **G**.
- OUTPUT: If **w** is in **L(G)**, a leftmost derivation of **w**; otherwise, an error indication.
- METHOD: Initially, the parser is in a configuration with **w$** in the input buffer and the start symbol **S** of **G** on top of the stack, above **$**. The following algorithm uses the predictive parsing table **M** to produce a predictive parse for the input.

```
let a be the first symbol of w;
let X be the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X = a ) pop the stack and let a be the next symbol of w;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1 Y2 ··· Yk ) {
                output the production X → Y1 Y2 ··· Yk;
                pop the stack;
                push Yk, Yk−1, ... , Y1 onto the stack, with Y1 on top;
        }
        let X be the top stack symbol;
}
```

Nonrecursive Predictive Parsing

Example 1: Consider the grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Nonrecursive Predictive Parsing

- Its Parsing Table is:

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

Moves made by a predictive parser on input

id + id * id

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

Example 2:

Consider the grammar

$S \rightarrow aBa$

$B \rightarrow bB \mid \varepsilon$

Construct Parsing table and parse for the string w = abba

- Example 2:

Consider the grammar

S → aBa

B → bB | ε

First(S) = {a}
First (B) = {b,ε}

Follow(S)= {$}
Follow(B)={a}

Parsing table will be:

	a	b	$
S	S → aBa		
B	B → ε	B → bB	

Parse for the string w = abba

- Given string $w =$ abba

Stack	Input	Action
$S	abba$	output S → aBa
$aBa	abba$	match **a**
$aB	bba$	output B → bB
$aBb	bba$	match **b**
$aB	ba$	output B → bB
$aBb	ba$	match **b**
$aB	a$	output B → ε
$a	a$	match **a**
$	$	Accept; Successful completion

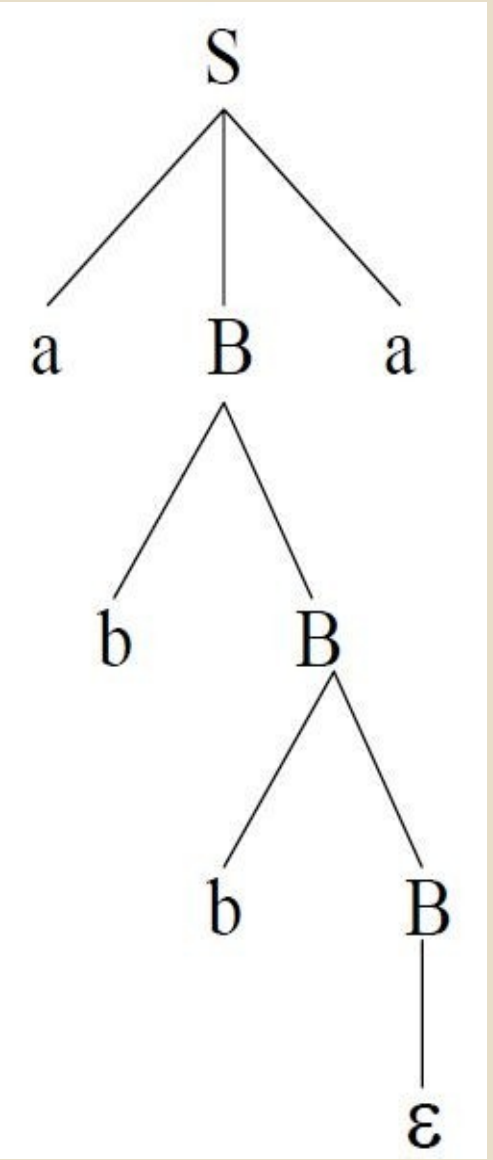
Output :-

$S \rightarrow aBa, B \rightarrow bB, B \rightarrow bB, B \rightarrow \varepsilon$

So, the leftmost derivation is

$S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$

And the parse tree will be:



LL(1) Grammars

- No ambiguous or left-recursive grammar is LL(1)
- There are no general rules by which multiple-defined entries can be made single-valued without affecting the language recognized by a grammar(i.e. there are no general rules to convert a non LL(1) grammar into a LL(1) grammar)

Consider the grammar:

S✉ iEtS | iEtSeS | a

E✉ b

Construct the non-recursive predictive parsing table for the given grammar.

Properties of LL(1) Grammar

- No Ambiguity and No Recursion
- In any LL(1) grammar, if there exists a rule of the form $A \rightarrow \alpha \mid \beta$, where α and β are distinct, then

1. For any terminal *a*, if $a \in$ FIRST(α) then $a \notin$ FIRST(β) or vice-versa
2. Either $\alpha \Rightarrow^* \varepsilon$ or $\beta \Rightarrow^* \varepsilon$ (or neither), but not both
3. If $\beta \Rightarrow^* \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A); likewise, if $\alpha \Rightarrow^* \varepsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A)

LL(1) Grammars

Grammar	**Not LL(1) Because**
$S \rightarrow S\ a \mid a$	Left Recursive
$S \rightarrow a\ S \mid a$	$FIRST(a\ S) \cap FIRST(a) \neq \varnothing$
$S \rightarrow a\ R \mid \varepsilon,\ R \rightarrow S \mid \varepsilon$	For R: $S \Rightarrow^* \varepsilon$ and $\varepsilon \Rightarrow^* \varepsilon$
$S \rightarrow a\ R\ a,\ R \rightarrow S \mid \varepsilon$	For R: $FIRST(S) \cap FOLLOW(R) \neq \varnothing$

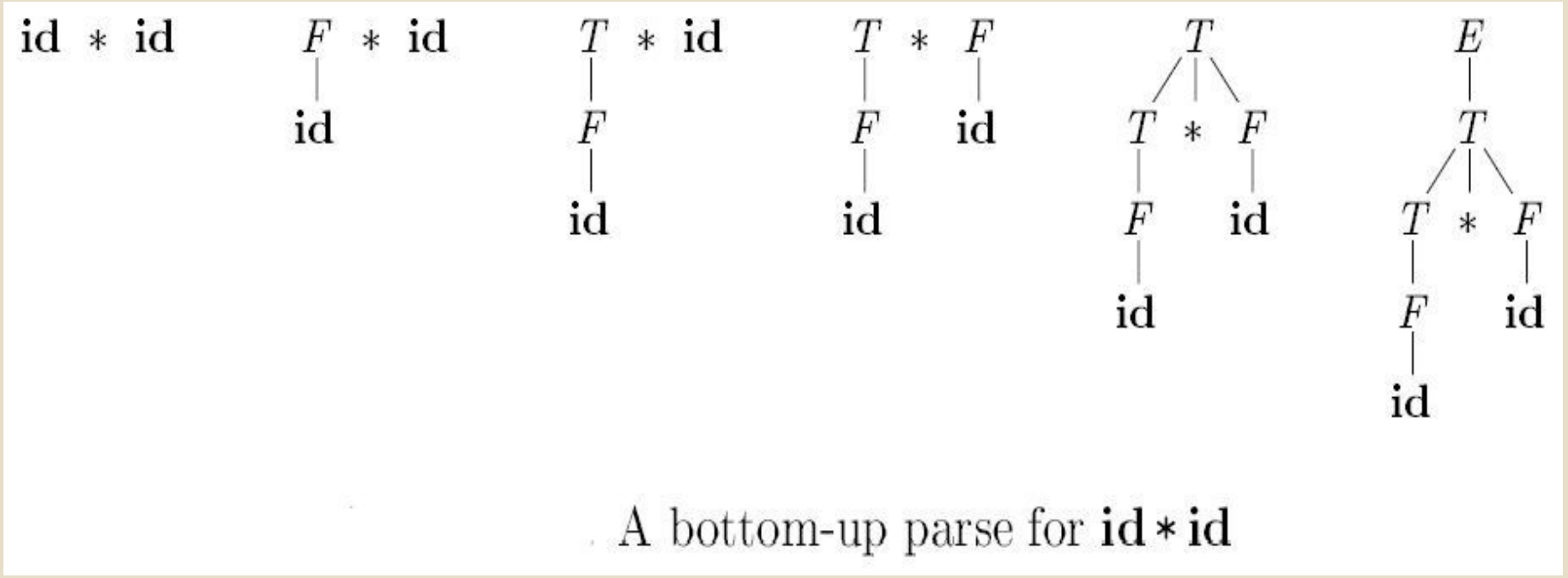
Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing due to following reasons
  - If the terminal symbol on the top of the stack does not match with the current input symbol
  - If the top of the stack is a non – terminal A, the current input symbol is *a*, and the entry for M[A, *a*] in parsing table is empty

- **What should the parser do in an error case?**
  - A parser should try to determine that an error has occurred as soon as possible. Waiting too long before declaring an error can cause the parser to lose the actual location of the error.
  - A suitable and comprehensive message should be reported.
  - After an error has occurred, the parser must pick a reasonable place to resume the parse. Rather than giving up at the first problem, a parser should always try to parse as much of the code as possible in order to find as many real errors as possible during a single run.

Bottom-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)



A bottom-up parse for **id * id**

Reductions

- We can think of bottom-up parsing as the process of "reducing" a string **w** to the start symbol of the grammar
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production
- The goal is to reduce all the way up to the start symbol and report a successful parse
- By definition, a reduction is the reverse of a step in a derivation

Reductions

- The goal of bottom-up parsing is therefore to construct a derivation in reverse

- The following corresponds to the parse shown above: $E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$

- This derivation is in fact a rightmost derivation

- Example(A)

Consider the grammar

S → aABe

A → Abc | b

B → d

Now, the sentence **abbcde** can be reduced to **S** as follows

abbcde

aAbcde (replacing b by using A → b)

aAde (replacing Abc by using A → Abc)

Handle

- 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
- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$

Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

Shift-Reduce Parsing

- A shift reduce parser tries to reduce the given input string into the starting symbol
- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by a non-terminal at the left side of that production rule
- If the substring is chosen correctly, the rightmost derivation of that string is created in reverse order
- For instance, in the above example (Example(A)):-

$$S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} aAbcde \underset{rm}{\Rightarrow} abbcde$$

Shift – Reduce Parser with Handle

Example

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

String : - id + id * id

Right – Most Sentential Form	Reduction Production	Handle
id + id * id	F → id	id
F + id * id	T → F	F
T + id * id	E → T	T
E + id * id	F → id	id
E + F * id	T → F	F
E + T * id	F → id	id
E + T * F	T → T * F	T * F
E + T	E → E + T	E + T
E		

Stack Implementation of Shift-Reduce Parser

- Like a table-driven predictive parser, a bottom-up parser makes use of a stack to keep track of the position in the parse and a parsing table to determine what to do next
- 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

Stack Implementation of Shift-Reduce Parser

- Algorithm

1. Initially stack contains only the sentinel $, and input buffer contains the input string w$
2. While stack is not equal to $S do
   - While there is no handle at the top of the stack, do shift input buffer and push the symbol onto the stack
   - If there is a handle on the top of the stack, then pop the handle and reduce the handle with its non-terminal and push it onto stack

Shift-Reduce Parser Actions

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

Stack Implementation of Shift-Reduc Parser

- Example Given grammar:

  E → E + T | T

  T → T * F | F F → (E) | id

String: **id + id * id**

Stack	Input	Action
$	id + id * id $	Shift id
$id	+ id * id $	Reduce by F → id
$F	+ id * id $	Reduce by T → F
$T	+ id * id $	Reduce by E → T
$E	+ id * id $	Shift +
$E +	id * id $	Shift id
$E + id	* id $	Reduce by F → id
$E + F	* id $	Reduce by T → F
$E + T	* id $	Shift * (OR Reduce by E → T) CONFLICT
$E + T *	id $	Shift id
$E + T * id	$	Reduce by F → id
$E + T * F	$	Reduce by T → T * F
$E + T	$	Reduce by E → E + T
$E	$	Accept

Conflicts in Shift-Reduce Parsing

- Some grammars cannot be parsed using shift-reduce parsing and result in conflicts
- There are two kinds of shift-reduce conflicts:
- **Shift/Reduce Conflict**
- Here, the parser is not able to decide whether to shift or to reduce
- Example: if A → ab|abcd, and the stack contains $ab, and the input buffer contains cd$, the parser cannot decide whether to reduce $ab to $A or to shift two more symbols before reducing

Conflicts in Shift-Reduce Parsing

- **Reduce/Reduce Conflict**
- Here, the parser cannot decide which sentential form to use for reduction
- For example if A → bc and B → abc and the stack contains $abc, the parser cannot decide whether to reduce it to $aA or to $B

LR-Parser

- The class of grammars called LR(k) grammars have the most efficient bottom-up parsers and can be implemented for almost any programming language
- The first L stands for left-to-right scan of the input buffer, the second R stands for a right-most derivation (leftmost reduction), and k stands for the maximum of look ahead
- If k is omitted, it is assumed to be 1

LR-Parser

- 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 LL parsers
- Virtually all programming language constructs for which CFGs can be written can be parsed with LR techniques
- The primary disadvantage is the amount of work it takes to build the tables by hand, which makes it infeasible to hand-code an LR parser for most grammars

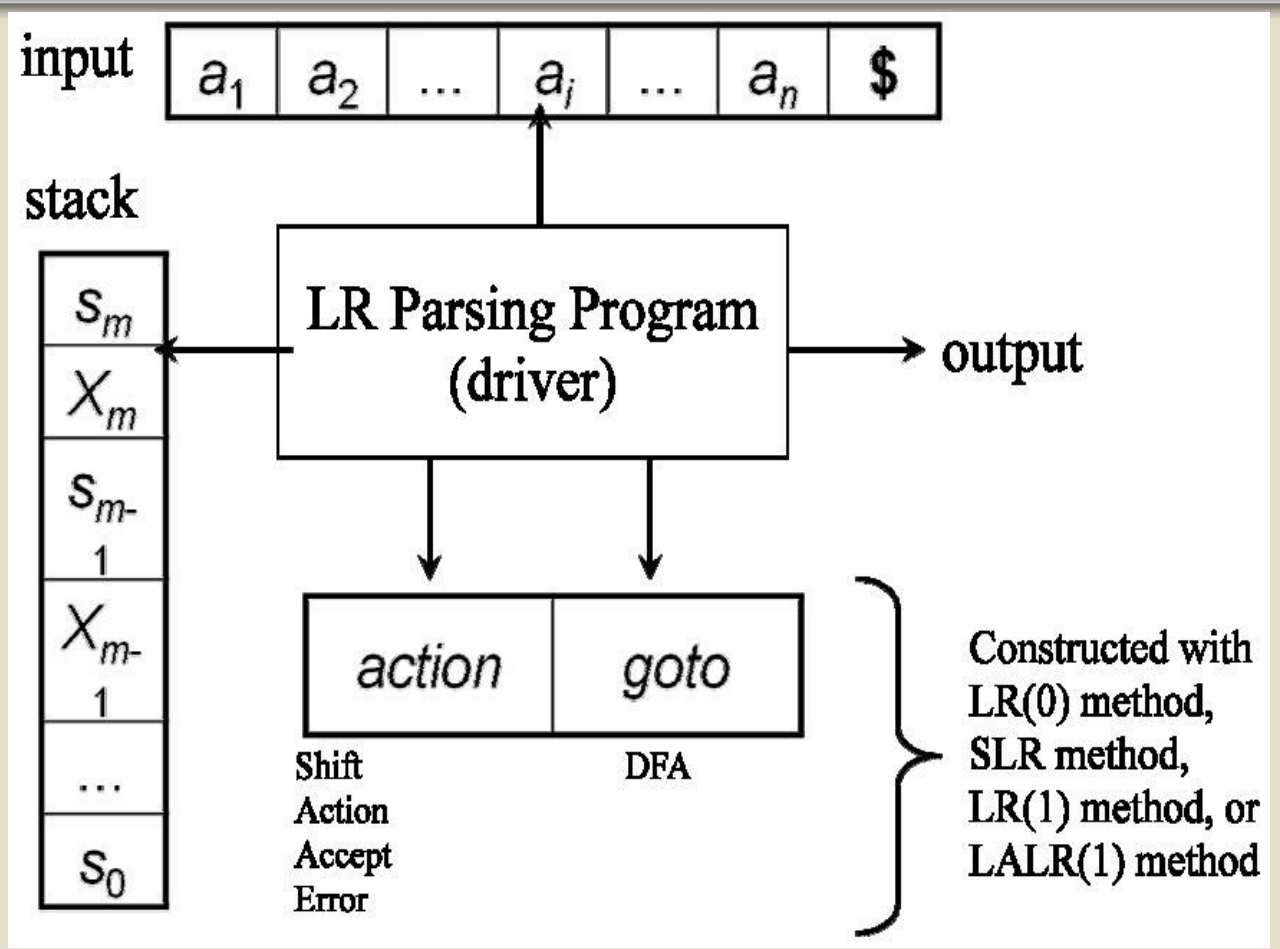
LR-Parser

- Fortunately, there are LR parser generators like Yacc(Bison) that create the parser from an unambiguous CFG specification
- The parser tool does all the tedious and complex work to build the necessary tables and can report any ambiguities or language constructs that interfere with the ability to parse it using LR techniques

LR-Parser

- LR-parsers cover wide range of grammars
  - SLR (Simple LR parser, LR(0))
  - LR (Most general LR parser)
  - LALR (Intermediate LR parser, Look ahead LR parser)
- All LR parsers use the same algorithm, the only difference is their parsing table

Structure of a General LR-Parser



Structure of a General LR-Parser

- An LR parser comprises of a stack, an input buffer and a parsing table that has two parts: **action** and **goto**
- Its stack comprises of entries of the form $s_0X_1\,s_1X_2:::X_ms_m$, where every s_i is called a "state" and every X_i is a grammar symbol (terminal or non-terminal)
- If top of stack is $\mathbf{s_m}$ and input symbol is **a**, the LR parser consults action $\mathbf{[s_m, a]}$ which can be one of four actions:

Structure of a General LR-Parser

1. shift s, where s is a state
2. reduce using production $A \rightarrow \beta$
3. accept
4. error

- The function goto takes a state and grammar symbol as arguments and produces a state
- The goto function forms the transition table of a DFA that recognizes the viable prefixes of the grammar

Structure of a General LR-Parser

- LR(k) grammars are less stringent than LL(k) grammars
- LR(k) grammars have to match the right side of a production by looking at its possible derivations with a maximum of k levels
- LL(k) grammars have to identify the right side of a production just by reading k terminals of the input string
- LR(k) grammars describe a larger class of grammars

LR Parser: Configurations/Actions

- The configuration of a LR parser is a tuple comprising of the stack contents and the contents of the unconsumed input buffer
- It is of the form ($s_0 X_1 s_1 \ldots X_m s_m$; $a_i a_{i+1} \ldots a_n$\$)

(1) If action [s_m, a_i] = shift s, the parser executes a shift move, entering the configuration ($s_0 X_1 s_1 \ldots X_m s_m a_i s$; $a_{i+1} \ldots a_n$\$) shifting both a_i and the new state s.

LR Parser: Configurations/Actions

(2)If action[s_m, a_i] = reduce $A \rightarrow \beta$, the parser executes a reduce move, entering the configuration ($s_0 X_1 s_1 \ldots\ldots X_{m-r} s_{m-r} A s$; $a_i a_{i+1} \ldots\ldots a_n \$$).

Here, s = goto[s_{m-r}, A] and r is the length of the handle β. (Note: If r is the length of β then the parser pops 2r symbols). After reduction, the parser outputs the production $A \rightarrow \beta$.

(3) If action[s_m, a_i] = accept then accept the grammar and stop.

(4) If action[s_m, a_i] = error then call error recovery routine.

Example

Grammar

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

$\Rightarrow$

state	action						goto		
	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			

Now, let the input string be id * id + id $

Stack	Input	Action	Output
$0	id * id + id $	M[0, id] = s5, so shift 5	
$0 id 5	* id + id $	M[5, *] = r6, reduce 6 goto 3, here goto 3 is because, the length of |id| = r = 1, and popping 2r means popping id and 5, there is state 0 and goto[0, F] = 3	F → id
$0 F 3	* id + id $	Reduce 4 goto 2	T → F
$0 T 2	* id + id $	Shift 7	
$0 T 2 * 7	id + id $	Shift 5	
$0 T 2 * 7 id 5	+ id $	Reduce 6 goto 10	F → id

$0 T 2 * 7 F 10	+ id $	Reduce 3 goto 2,here r = |T*F| = 3 and pop 2r = 6, popping 6 element we got 0 and goto[0, T] = 2	T → T * F
$ 0 T 2	+ id $	Reduce 2 goto 1	E → T
$0 E 1	+ id $	Shift 6	
$0 E 1 + 6	id $	Shift 5	
$0 E 1 + 6 id 5	$	Reduce 6 goto 3	F → id
$0 E 1 + 6 F 3	$	Reduce 4 goto 9	T → F
$0 E 1 + 6 T 9	$	Reduce 1 goto 1	E → E + T
$0 E 1	$	Accept	

Constructing the Parsing Table : SLR Parsing Table

- SLR parsers are the simplest class of LR parsers
- Constructing a parsing table for action and goto involves building a state machine that can identify handles
- For building a state machine, we need to define three terms: item, closure and goto

Item sets

- An “item” (LR(0) item) is a production rule that contains a dot (•) somewhere in the right side of the production
- For example, the production $A \rightarrow \alpha A \beta$ has four items:

$A \rightarrow \bullet \alpha A \beta$

$A \rightarrow \alpha \bullet A \beta$

$A \rightarrow \alpha A \bullet \beta$

$A \rightarrow \alpha A \beta \bullet$

Item sets

- The production $A \rightarrow \varepsilon$, generates only one item $A \rightarrow \bullet$
- An item indicates how much of a production we have seen at a given point in the parsing process
- For example, the first item above indicates that we hope a string derivable from $\alpha A \beta$ next on the input
- The second item indicates that we have just seen on a input string derivable from α, and that we hope next to see a string derivable from $A\beta$

Item sets

- An item encapsulates what we have read until now and what we expect to read further from the input buffer
- Sets of LR(0) items will be the states of action and goto table of the SLR parser
- A collection of sets of LR(0) items (the canonical LR(0) collection) is the basis of constructing SLR parsers
- To construct the canonical LR(0) collection we require augmented grammar and two functions **closure** and **goto**

Closure Operation

- If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I using the following rules:
  - Initially, every item in I is added to closure(I).
  - If $A \rightarrow \alpha \bullet B\beta$ is in closure (I), and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \bullet\gamma$ to I if it is not already there. Apply this rule until no new items can be added to closure(I).

Example

Consider a grammar:

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

If $I_0 = \{[E' \rightarrow \bullet E]\}$, then closure(I) contains the items:

$E' \rightarrow \bullet E$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet id$

goto Operation

- In any item I, for all productions of the form $A \rightarrow \alpha \bullet X\beta$ that are in I, goto[I, X] is defined as the closure of all productions of the form $A \rightarrow \alpha X \bullet \beta$
- Example
- In above case, I_0 = closure({[$E' \rightarrow \bullet E$]}),

then goto[I_0, E] = I_1

where,

I_1 = closure({[$E' \rightarrow E\bullet$], [$E \rightarrow E\bullet+T$]})

Algorithm for Constructing SLR Parsing Table

1. Given the grammar G, construct a an augmented grammar G' by introducing a production of the form S' → S, where S is the star symbol of G.
2. Let I_0 = *closure*({[S' → *S*]}). Starting from I_0, construct SLR DFA using *closure* and *goto*.
3. State *i* is constructed from I_i. The parsing actions for state *i* are defined as follows:
4. If *goto*[I_i, A] = I_j , where A is a non-terminal, then set *goto*[*i*, A] = *j*
    a. If *goto*[I_i, A] = I_j , set *action*[*i*, a] = shift *j*, here *a* must be a teminal
    b. If I_i has the production of the form A → α •, then for all symbols a in FOLLOW(α), set *action*[*i*, a] = reduce A → α •. Here A should not be S'.
    c. If S' → *S* • is in I_i, then set *action*[*i*, $] = ***accept***
5. For all blank entries (i.e. *action* and *goto* entries not defined by steps 2 and 3), set entries to "***error***".
6. The start state s_0 corresponds to I_0 = *closure*({[S' → *S*]}).

Example

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Augmented Grammar is:

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Now,

I_0 = closure ([E′ → •E]) = E′ → •E (here, after dot(•), there is non-terminal so expand it) E → •E + T

E → •T (again, there is dot(•) after T so expand the production of T) T → •T * F

T → •F (again do the same for F)

F → •(E)

F → •id (no non-terminal remain after dot(•), so go for I_1)

Now for every terminals and non-terminals after dot(•), find goto

$I_1 = goto(I_0, E) = E' \rightarrow E\bullet$

$E \rightarrow E\bullet + T$

$I_2 = goto(I_0, T) = E \rightarrow T\bullet$

$T \rightarrow T\bullet * F$

$I_3 = goto(I_0, F) = T \rightarrow F\bullet$

$I_4 = goto(I_0, (\,)) = F \rightarrow (\bullet E)$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet id$

$I_5 = goto(I_0, id) = F \rightarrow id\bullet$

Now, goto for I_0 is finished, now look for every members in I_1, that has dot(•) before it, and use goto, and continue so on.

$I_6 = goto(I_1, +) = E \rightarrow E+\bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet id$

$I_7 = got(I_2, *) = \quad T \rightarrow T * \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet id$

No possible goto for I_3 since there is not any member after dot(•)

$I_8 = \text{goto}(I_4, E) = F \rightarrow (E\bullet)$

$E \rightarrow E\bullet + T$

$I_9 = \text{goto}(I_4, T) = E \rightarrow T\bullet$

$T \rightarrow T\bullet * F$

Which is same as I_2, so no new state?

Similarly, $\text{goto}(I_4, T) = I_2$, $\text{goto}(I_4, F) = I_3$, $\text{goto}(I_4, id) = I_5$, $\text{goto}(I_4, ( ) = I_4$

Since no goto possible for I_5, go for I_6

$I_9 = \text{goto}(I_6, T) = E \rightarrow E+T\bullet$

$$T \rightarrow T\bullet * F$$

Since, goto(I_6, F) = I_3, goto(I_6, ( ) = I_4, goto(I_6, id) = I_5, go for others new state

I_{10} = goto(I_7, F) = $T \rightarrow T * F\bullet$

Since, goto(I_7, ( ) = I_4, goto(I_6, id) = I_5, go for others new state

I_{11} = goto(I_8,)) = $F \rightarrow (E)\bullet$

goto(I_8, +) = I_6, goto(I_9, *) = I_7

Now, finished stop and total numbers of state = 12

Rules:

1. If $[A \rightarrow \alpha \bullet a\beta]$ is in I_i and **goto**$[I_i, a] = I_j$, then **action**$[I_i, a] = s_j$, here **a** is a terminal
2. If $[A \rightarrow \alpha \bullet]$ is in I_i, then **action**$[I_i, a]$ = reduce $A \rightarrow \alpha$ for all **a** in FOLLOW(A), here A is not S′
3. If $[S' \rightarrow S \bullet]$ is in I_i, then **action**$[I_i, \$]$ = accept

Now we build the SLR parsing table:

State	action						goto		
	id	+	*	(	)	$	E	T	F

Now, among all I_i, let us chose those productions which match either $A \rightarrow \alpha \bullet a\beta$, or $A \rightarrow \alpha \bullet$, or $S' \rightarrow S$. For I_0, the production matching in our rules are only $F \rightarrow \bullet (E)$ and $F \rightarrow \bullet id$

The first, $F \rightarrow \bullet (E)$, matches the production $A \rightarrow \alpha \bullet a\beta$, so apply rule 1. i.e. here $goto[I_0, (\ ] = I_4$, that means j = 4, so shift with 4, here the terminal value of ***a*** is (.

Similar case for second production, shift with 5, since $goto[I_0, id\ ) = I_5$,

Also, $goto[I_0, E\ ] = I_1$, $goto[I_0, T\ ] = I_2$, $goto[I_0, F\ ] = I_3$,

Now the parsing table looks like

State	**action**						**goto**		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3

For I_1, two productions E' → E• and E → E•+T matches, where first followed Rule 3, since E is the starting symbol in our grammar, while next follow Rule 1.

State	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			

For I_2, the matching productions are E → T•, T → T• * F. The second production works as above, and *goto*[I_2, *] = I_7, shift with 7. For first production, we have to apple Rule 2. As comparing with A → α•, we have to calculate FOLLOW(A) = FOLLOW(E) = {+,), $}. Now, *action*[$I_2$, +] = *action*[$I_2$,)] = *action*[$I_2$, $] = reduce by production E → T , which is production number 2 in our grammar, so table looks like now

State	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r2	s7		r2	r2			

Now as continuing the same process, we get the following final table:

STATE	ACTION						GOTO		
	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 and Ambiguity (Limitation of SLR)

- Every SLR grammar is unambiguous
- But there exist certain unambiguous grammars that are not SLR
- In such grammar, there exists at least one multiple defined entry **action[i, a]**, which contains both a shift and a reduce directive
- When we have a completed configuration (i.e., dot at the end) such as

  $A \rightarrow \alpha\bullet$, we know that this corresponds to a situation in which we have α as a handle on top of the stack which we then can reduce, i.e., replacing α by A

- We allow such a reduction whenever the next symbol is in FOLLOW(A)
- However, it may be that we should not reduce for every symbol in FOLLOW(A), because the symbols below α on the stack preclude being a handle for reduction in this case
- In other words, SLR states only tell us about the sequence on top of the stack, not what is below it on the stack
- We may need to divide an SLR state into separate states to differentiate the possible means by which that sequence has appeared on the stack
- By carrying more information in the state, it will allow us to rule out these invalid reductions

Example: Consider a grammar:

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow * R$

$L \rightarrow id$

$R \rightarrow L$

Compute the SLR parsing table.

For this grammar, we would get the following SLR parsing table

state	**action**				**goto**		
	id	=	*	$	S	L	R
0	s5		s4		1	2	3
1				accept			
2		s6/r5		r5			
3				r2			
4	s5		s4			8	7
5		r4		r4			
6			s4	s5		8	9
state	**action**				**goto**		
	id	=	*	$	S	L	R
7		r3		r3			
8		r5		r5			
9				r1			

- In state 2, action[2, =] = s6 and action[2, =] = r5, it is seen that there is shift-reduce conflict
- Suppose the input string was of the form id = ….., where id was reduced to L and "=" is the next input symbol, and the state of the parser is in I_2
- Because R → L• is contained in I_2, the parser tries to reduce L to R according to the reduction rule
- The sentential form now becomes R = …..
- But we see that no right hand side of a rule has R = ….., anywhere in the grammar
- To remove such ambiguity we use LR(1) grammar

LR(1) Grammar

- It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by $A \rightarrow \alpha$
- The extra information is incorporated into the state by redefining items to include a terminal symbol as a second component
- The general form of an item is of the form $[A \rightarrow \alpha \bullet \beta, a]$ where $A \rightarrow \alpha\beta$ is a production and **a** is a terminal or the right end marker $
- We call such an object an LR(1) item

LR(1) Grammar

- In an item of the form [A → α•, a], reduction is performed using A → α only if the next input symbol is **a**
- The input symbol **a** is called the “lookahead” (which is of length 1)
- LR(1) item = LR(0) item + lookahead
- The parsers that uses LR(1) items is also called Canonical-LR

Computation of Closure(I) for LR(1) Items

1. Repeat
2. For each item of the form [A → α•Bβ, a] in I

   For each production of the form B → γ in G′

   For each terminal b in FIRST(βa)

   add [B → •γ, b] to I if it is not already there
3. Until no more items can be added to I

Computation of GOTO(I, X) for LR(1) Items

For each item ofthe form $[A \rightarrow \alpha \bullet X\beta, a]$ in I

goto[I, X] = Closure({$[A \rightarrow \alpha X \bullet \beta, a]$})

Construction of Canonical LR(1) Parsing Table

1. Given the grammar G, construct an augmented grammar G′ by introducing a production of the form $S' \rightarrow S$, where S is the start symbol of G
2. Construct the set $C = \{I_0, I_1, \ldots\ldots, I_n\}$ of LR(1) items
3. If $[A \rightarrow \alpha \bullet a\beta, b] \in I_i$ and **goto(I_i, a)** = I_j then set **action[i, a]** = $shift_j$
4. If $[A \rightarrow \alpha \bullet, b] \in I_i$ then set **action[i, b]** = reduce $A \rightarrow \alpha$ (apply only if $A \neq S'$)
5. If $[S' \rightarrow S \bullet, \$]$ is in I_i then set **action[i, $]** = accept
6. If **goto(I_i, A)** = I_j then set **goto[i, A]** = j
7. Repeat 3 – 6 until no more entries are added
8. The initial state I is the I_i holding item $[S' \rightarrow \bullet S, \$]$

- Example

Consider the grammar

$S \rightarrow CC$

$C \rightarrow eC$

$C \rightarrow d$

Augment the grammar

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow eC$

$C \rightarrow d$

Find I_0 = Closure (S' → S, $)

S' → •S, $

Comparing it with A → α •Bβ, a, we get A =**S'**, α = ε, S = B, β = ε, a = $. Now, FIRST($\beta$a) = FIRST($) = $

Now apply the productions of B = S, with lookahead FIRST(βa) = $, That means

S → •CC, $

Now again, comparing it with A → α •Bβ, a, we get A =**S**, α = ε, S = C, β = C, a = $. Now, FIRST($\beta$a) = FIRST(C$) = FIRST(C) = {e, d}. Now

C →•eC, e

C →•eC, d

C →•d, e

C →•d, d

We can rewrite this as C → eC, e / d and C → d, e / d

So, $I_0 = S' \rightarrow \bullet S, \$$

$S \rightarrow \bullet CC, \$$

$C \rightarrow \bullet eC, e / d$

$C \rightarrow \bullet d, e / d$

$I_1 = goto(I_0, S) = S' \rightarrow S\bullet, \$$

$I_2 = goto(I_0, C) = S \rightarrow C\bullet C, \$$

$C \rightarrow \bullet eC, \$$

$C \rightarrow \bullet d, \$$

$I_3 = goto(I_0, e) = C \rightarrow e\bullet C, e / d$

$C \rightarrow \bullet eC, e / d$

$C \rightarrow \bullet d, e / d$

$I_4 = goto(I_0, d) = C \rightarrow d\bullet, e / d$

$I_5 = goto(I_2, C) = S \rightarrow CC\bullet, \$$

$I_6 = goto(I_2, e) = C \rightarrow e\bullet C, \$$

$C \rightarrow \bullet eC, \$$

$C \rightarrow \bullet d, \$$

$I_7 = \text{goto}(I_2, d) = C \rightarrow d\bullet, \$$

$I_8 = \text{goto}(I_3, C) = C \rightarrow eC\bullet, e / d$

$\text{goto}(I_3, e) = I_3$

$\text{goto}(I_3, d) = I_4$

$I_9 = \text{goto}(I_6, C) = C \rightarrow eC\bullet, \$$

$\text{goto}(I_6, e) = I_6$

$\text{goto}(I_6, d) = I_7$

Now, let us build the canonical LR(1) parsing table.

For I_0, for production rule {C → •eC, e/d}, a = e, and goto[I_0, e] = I_3,
From Rule 3, **action[0, e]** = shift 3
Similarly, for {C → •d, e/d}, **action[0, d]** = shift 4

For I_4, {C → d•, e/d}, b = {e, d}, so applying rule 4, **action[4, e]** = **action[4, d]** = reduce by production C → d

After performing all computations, we get the following parsing table:

state	action			goto	
	e	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

LALR Parsing

- Because a canonical LR(1) parser splits states based on differing lookahead sets, it can have many more states than corresponding SLR parser
- Potentially, it could require splitting a state with just one item into a different state for each subset of the possible lookaheads
- With LALR (lookahead LR) parsing, we attempt to reduce the number of states in an LR(1) parser by merging similar states

- This reduces the number of states to the same as SLR, but still retains some of the power of the LR(1) lookaheads
- LALR grammars are midway between SLR (simplest) and LR (most complex) grammar
- They perform generalization over canonical LR item sets
- A typical programming language generates thousands of states for canonical LR parsers, while they generate only hundred of states for SLR and LALR parsers
- LALR parsers suffice for most programming languages' requirements

Union Operation on Items

- Given an item of the form $[A \rightarrow \alpha \bullet B\beta, a]$, the first part (before comma) is called the "core" of the item
- Given two states of the form $I_i = \{[A \rightarrow \alpha \bullet, a]\}$ and $I_j = \{[A \rightarrow \alpha \bullet, b]\}$, the union of the states $I_{ij} = I_i \cup I_j = \{[A \rightarrow \alpha \bullet, a / b]\}$
- I_{ij} will perform a reduce operation on seeing either **a** or **b** on the input buffer, whereas I_i and I_j are more stringent
- The state machine resulting from the above union operation has one less state

Union Operation on Items

- If I_i and I_j have more than one item, then the set of all core elements in I_i should be the same as the set of all core elements in I_j for the union operation to be possible
- Union operations do not create any new shift-reduce conflicts, but can create new reduce-reduce conflicts
- Shift operation depends only on the core and not on the next input symbol

Union Operation on Items

- We may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser

I_1 : $A \rightarrow \alpha\bullet, a$

$B \rightarrow \beta\bullet, b$

I_2 : $A \rightarrow \alpha\bullet, b$

$B \rightarrow \beta\bullet, c$

I_{12} : $A \rightarrow \alpha\bullet, a$

$A \rightarrow \alpha\bullet, b$

$B \rightarrow \beta\bullet, b$

$B \rightarrow \beta\bullet, c$

reduce/reduce conflict in case of

$A \rightarrow \alpha\bullet, b$ and

$B \rightarrow \beta\bullet, b$

LALR Parsing Table Construction

1. Construct C = $\{I_0, I_1, \ldots\ldots, I_n\}$, the collection of sets of LR(1) items.

2. For each core present in C, find all sets having that core, and replace these sets by their union. Let C′ = $\{J_0, J_1, \ldots\ldots, J_m\}$ be the resulting set of LR(1) items.

3. Construct the **action** elements of parsing table using the same method as for canonical LR(1) grammars.

4. If J is the union of **k** LR(1) items, J = $I_1 \cup I_2 \cup \ldots\ldots \cup I_k$, then the cores of goto[I_1, X], goto[I_2, X], etc. are all same since $I_1, I_2, \ldots\ldots, I_k$ have the same core. Let K be the union of all sets of items having the same core as goto[I_1, X]. Then goto[J, X] = K

Example:

Consider the grammar that we used for LR(1) parsing

$S \rightarrow CC$

$C \rightarrow eC$

$C \rightarrow d$

As we have seen, there are 10 item sets I_0 to

Among them I_3 and I_6, I_4 and I_7, I_8 and I_9 are equivalent because their core items are same

I_3	I_6	I_4	I_7	I_8	I_9
C → e•C, e / d C → •eC, e / d C → •d, e / d	C → e•C, $ C → •eC, $ C → •d, $	C → d•, e/d	C → d•, $	C → eC•, e/d	C → eC•,$

The corresponding LR(1) parsing table for this grammar was

LR(1) Parsing Table					
state	**Action**			**goto**	
	e	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Now the equivalent LALR parsing table looks like

LALR Parsing Table					
state	**Action**			**goto**	
	e	**d**	$	S	C
0	s36	s47		1	2
1			accept		
2	s36	s47			5
3, 6	s36	s47			89
4, 7	r3	r3	r3		
5			r1		
8, 9	r2	r2	r2		

How GOTO is computed:

Consider GOTO(I_{36}, C). In the set of LR(1) items, GOTO(I_3, C) = I_8, and I_8 is now part of I_{89}, so we make GOTO(I_{36}, C) be I_{89}. The cases for I_{36} and I_{47} is similar.

For another example, consider (I_2, e), an entry that leads to the shift action of I_2 on input e. In the set of LR(1) items, GOTO(I_2, e) = I_6. Since I_6 is now part of I_{36}, GOTO(I_2, e) becomes I_{36}. Thus, the entry for state 2 in above figure and input e is made s_{36}, meaning push state 36 onto the stack.

Kernel and Non – Kernel Items

- We can represent any set of LR(0) or LR(1) items **I** by its kernel, that is, by those items that are either the initial item – [S′ → •S] or [S′ → •S, $]
  – or that have the dot somewhere other than at the beginning of the production body
- Other than the initial item, no other item generated by a **goto** has a dot at the left end of the production
- Items that are generated by closure over kernel items have a dot at the beginning of the production and these items are called non-kernel items

Example

Consider the grammar:

$$
\begin{array}{lcl}
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
$$

The canonical collection of sets of LR(0) items for this grammar is:

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S\cdot$

I_2: $S \to L\cdot = R$
$R \to L\cdot$

I_3: $S \to R\cdot$

I_4: $L \to *\cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id}\cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_7: $L \to *R\cdot$

I_8: $R \to L\cdot$

I_9: $S \to L = R\cdot$

The kernels of these items are:

I_0: $S' \to \cdot S$

I_1: $S' \to S\cdot$

I_2: $S \to L\cdot = R$
$R \to L\cdot$

I_3: $S \to R\cdot$

I_4: $L \to *\cdot R$

I_5: $L \to \mathbf{id}\cdot$

I_6: $S \to L = \cdot R$

I_7: $L \to *R\cdot$

I_8: $R \to L\cdot$

I_9: $S \to L = R\cdot$