COMP409: COMPILER DESIGN

3. SYNTAX ANALYSIS

Instructor: Sushil Nepal, Assistant Professor

Block: 9-308

Email: sushilnepal@ku.edu.np

Contact: 9851-151617

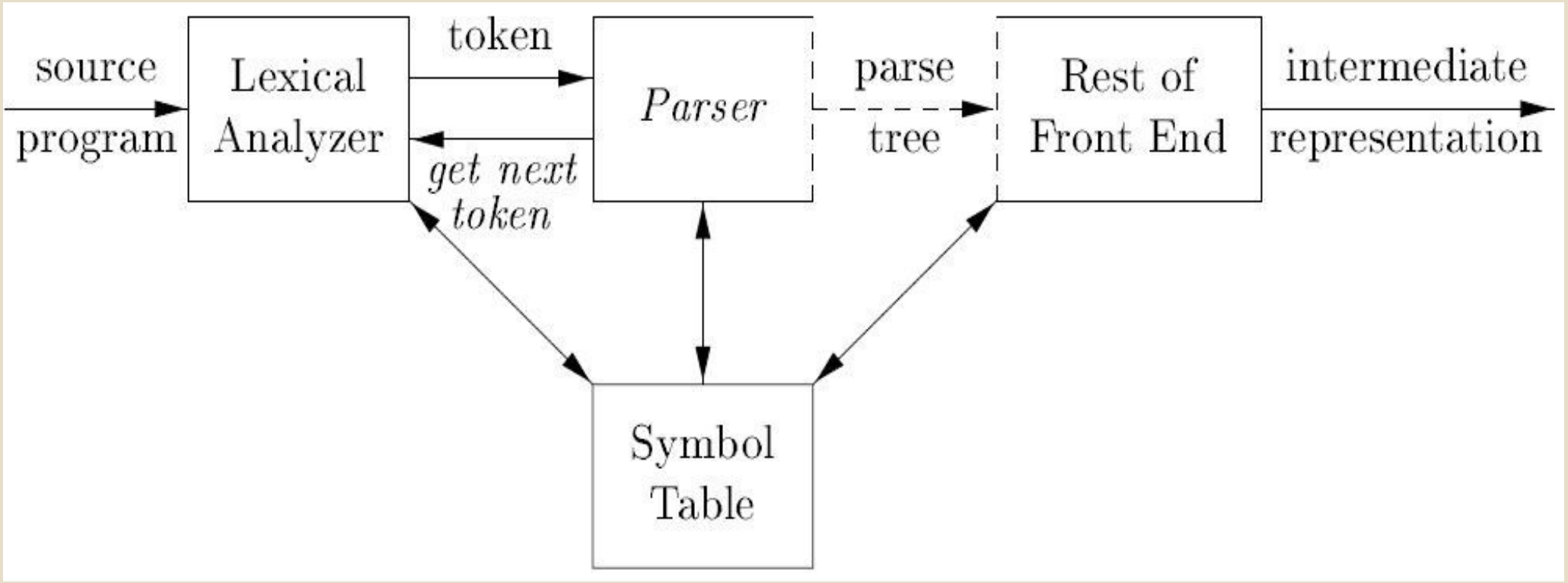
Introduction

- 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:

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

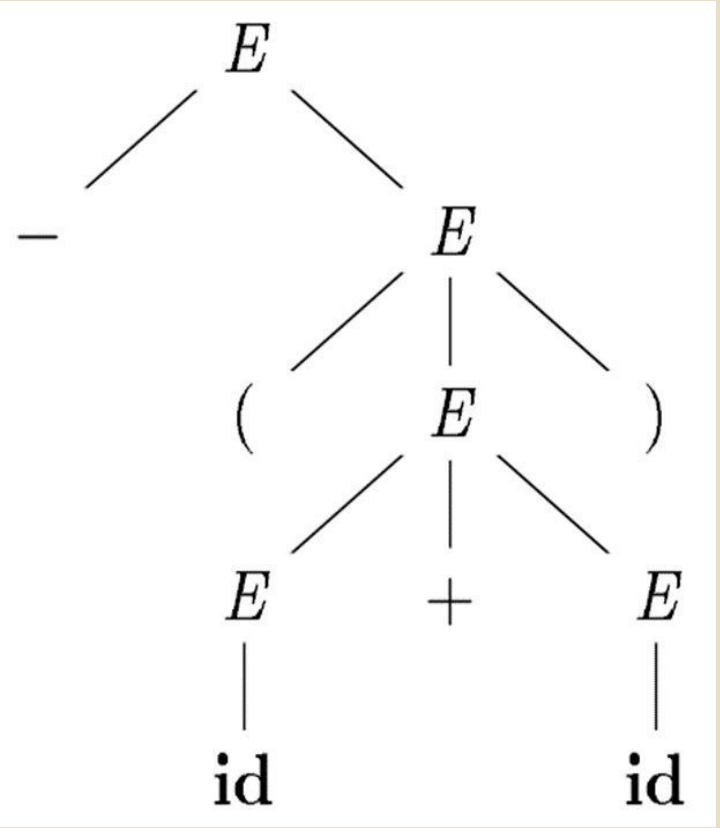
- The string -(id + id) is a sentence of this grammar because there is a derivation:

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

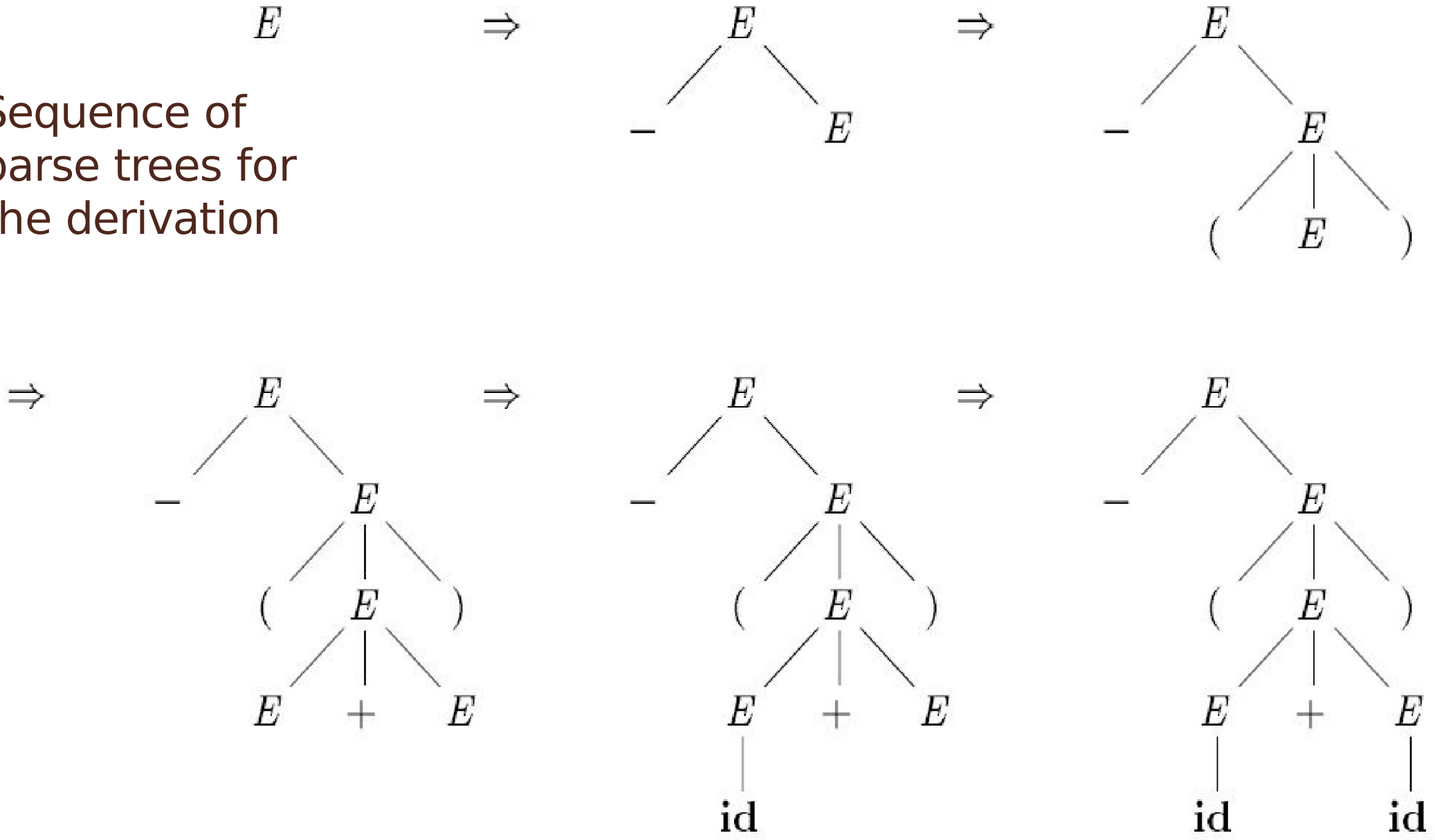
Parse Tree

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:

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

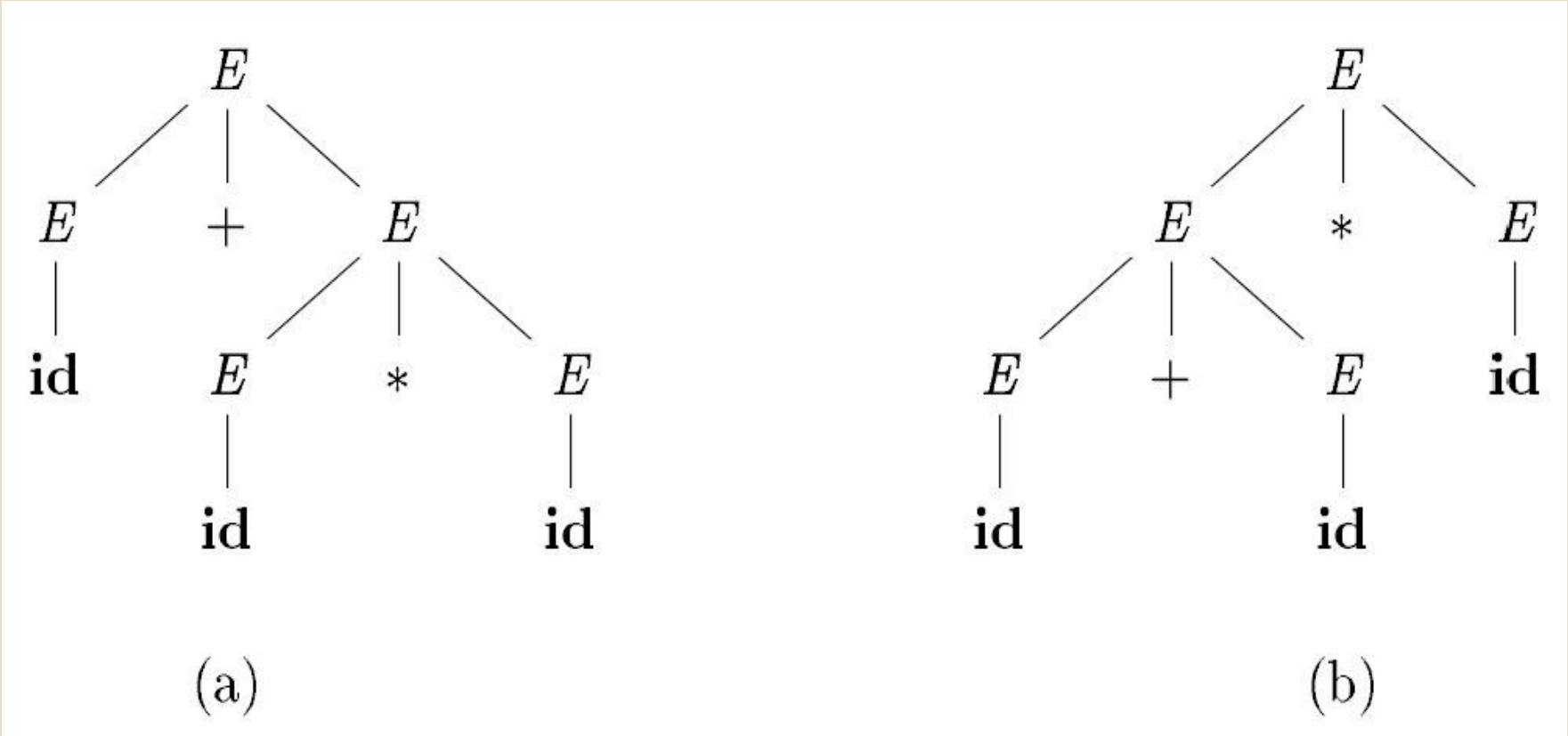
- Consider the string: **id + id * id**
- It has two distinct derivations:

(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.

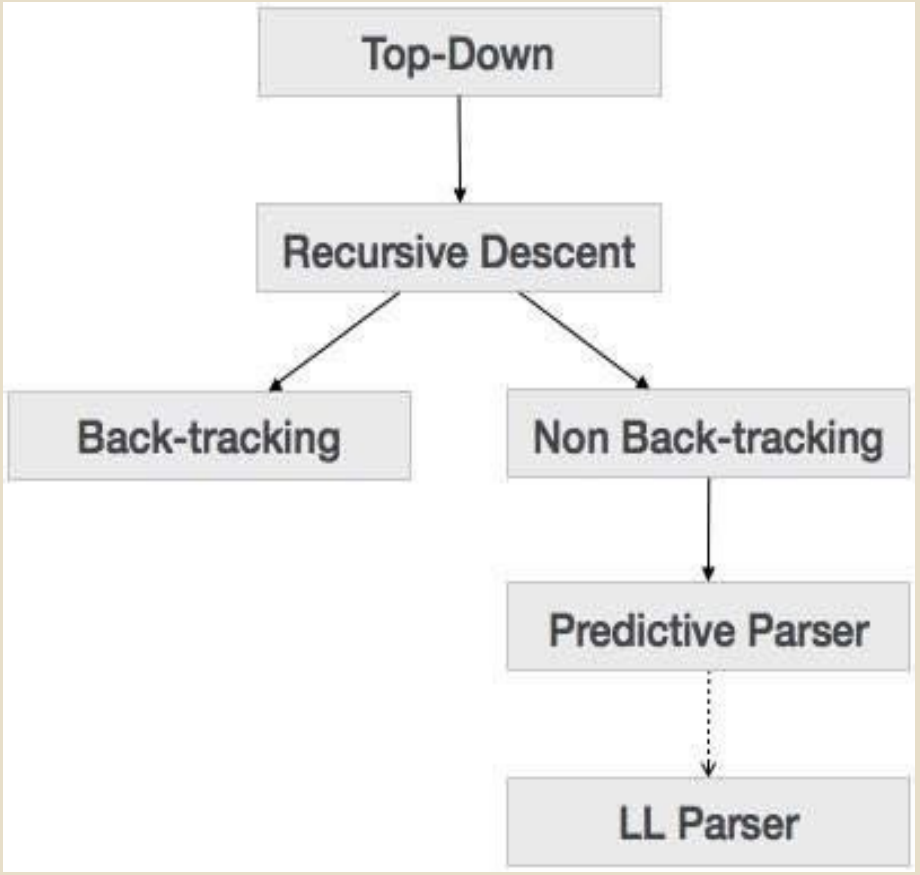
Top-Down Parsing

- 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

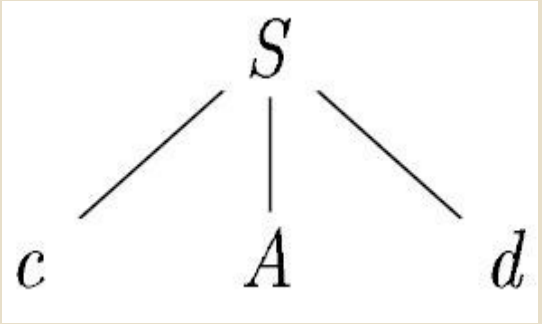
- Consider the grammar

  $S \rightarrow cAd$

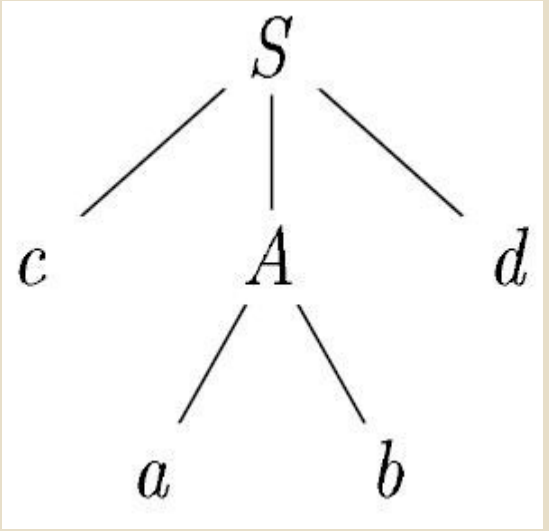
  $A \rightarrow ab|a$

- 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:

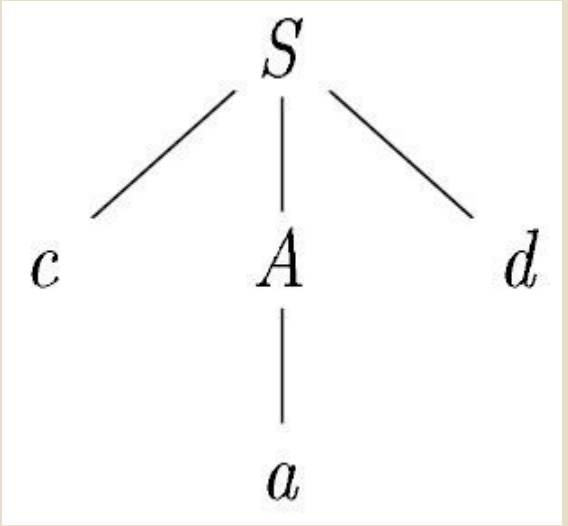


- The leftmost leaf, labeled **c**, matches the first symbol of input w, so we advance the input pointer to **a**, the second symbol of w, and consider the next leaf, labeled A
- Now, we expand A using the first alternative A → ab to obtain the subsequent tree



- 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:

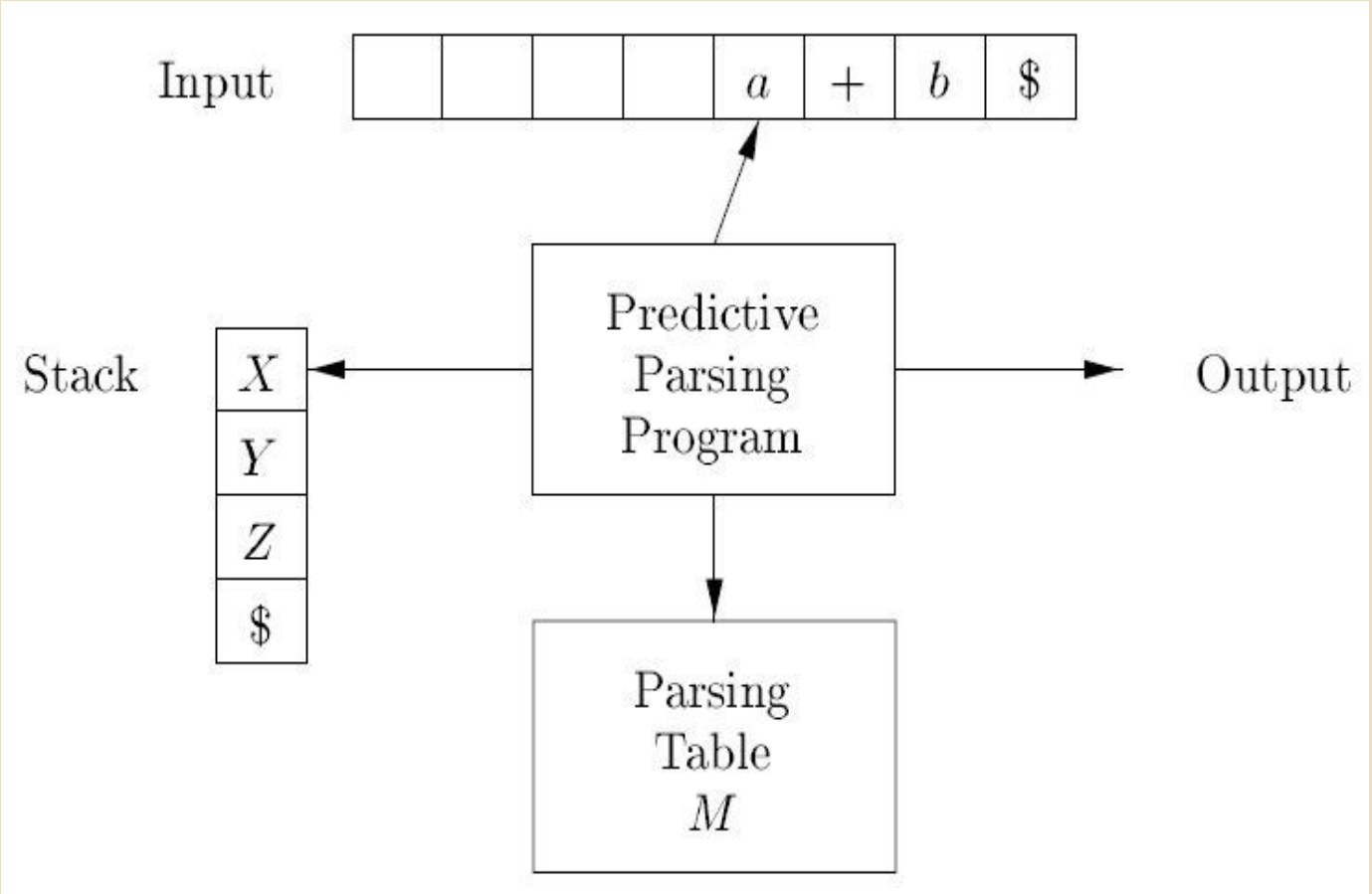
S → cAd

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 → aBa

B → bB | ε

Parsing table will be:

	a	b	$
S	S → aBa		
B	B → ε	B → bB	

- 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

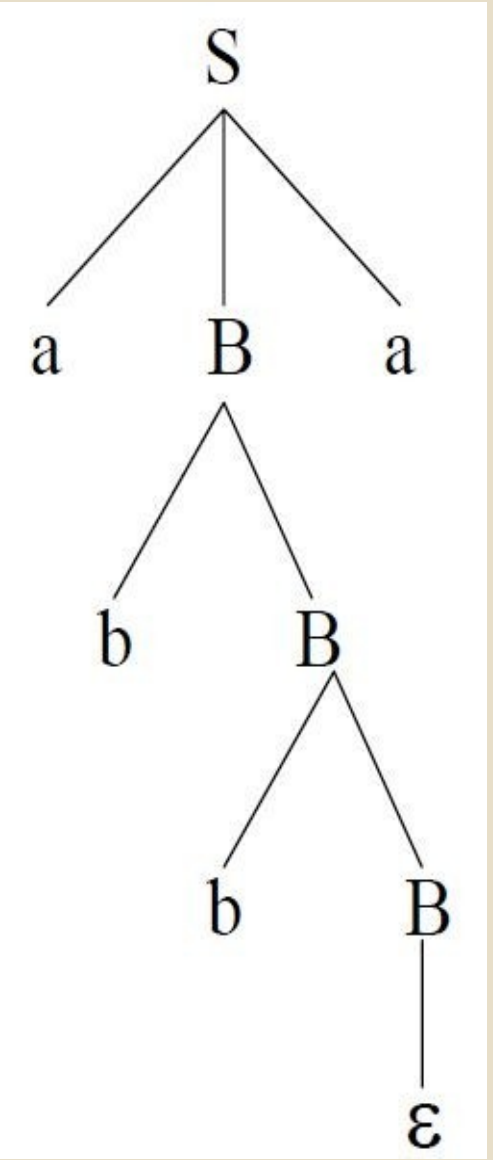
Output :-

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

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)

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.