SYNTAX ANALYSIS

Top-down parsing

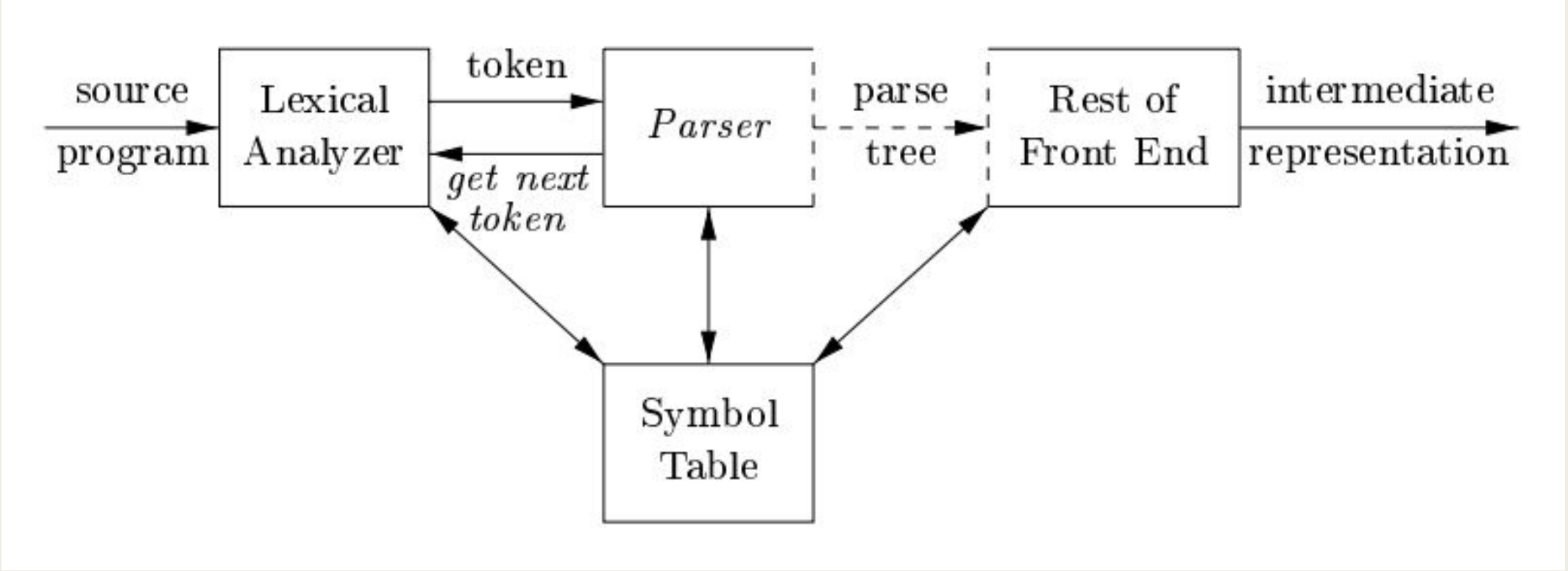
Syntax of programming languages

- By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs.
- In C, for example, a program is made up of functions, a function out of declarations and statements, a statement out of expressions, and so on.
- The syntax of programming language constructs can be specified by context-free grammars or BNF (Backus-Naur Form) notation.

The Role of the Parser

- The parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language.
- Our aim is to design the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program.
- For well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.
- In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing, as we shall see.

The Role of the Parser



Types of Parsers

- There are three general types of parsers for grammars: universal, top-down, and bottom-up.
  - *Many universal methods can parse any grammar but these general methods are too inefficient to use in production compilers.*
- The methods commonly used in compilers can be classified as being either top-down or bottom-up.

Error Handling

- If a compiler had to process only correct programs, its design and implementation would be simplified greatly. However, a compiler is expected to assist the programmer in locating and tracking down errors that inevitably creep into programs, despite the programmer's best efforts.
- Common programming errors can occur at many different levels.
  - **Lexical errors** include misspellings of identifiers, keywords, or operators.
  - **Syntactic errors** include misplaced semicolons or extra or missing braces; that is, "{" or "}"
  - **Semantic errors** include type mismatches between operators and operands.
  - **Logical errors** can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==.

Error Handling

- The error handler in a parser has goals that are simple to state but challenging to realize:
  - *Report the presence of errors clearly and accurately.*
  - *Recover from each error quickly enough to detect subsequent errors.*
  - *Add minimal overhead to the processing of correct programs.*
- Once an error is detected, what should the parser do? The simplest approach is for the parser to quit with an informative error message when it detects the first error. Additional errors are often uncovered if the parser can restore itself to a state where processing of the input can continue with reasonable hopes that the further processing will provide meaningful diagnostic information.

Error Recovery Strategies

- **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. The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source program is clear and unambiguous. While panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has the advantage of simplicity.
- **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. The choice of the local correction is left to the compiler designer. Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string. Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

Error Recovery Strategies

- **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.
- **Global Correction:** Ideally, 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. 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 Grammars

- A context-free grammar (grammar for short) consists of terminals, nonterminals, a start symbol, and productions.
- Terminals are the basic symbols from which strings are formed.
- Nonterminals are syntactic variables that denote sets of strings. The sets of strings denoted by nonterminals help define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.
- In a grammar, one nonterminal is distinguished as the start symbol, and the set of strings it denotes is the language generated by the grammar.

Context-Free Grammars

- The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each production consists of:
  - A nonterminal called the head or left side of the production.
  - The symbol →. Sometimes ::= has been used in place of the arrow.
  - A body or right side consisting of zero or more terminals and nonterminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

Context-free Grammar

In this grammar, the terminal symbols are

id + - * / ( )

The nonterminal symbols are *expression*, *term* and *factor*, and *expression* is the start symbol □

expression	→	*expression* + *term*
expression	→	*expression* - *term*
expression	→	*term*
term	→	*term* * *factor*
term	→	*term* / *factor*
term	→	*factor*
factor	→	(*expression*)
factor	→	**id**

Derivations

- You use a grammar to describe a language by generating each string of that language in the following manner.
  - *Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise.*
  - *Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.*
  - *Repeat step 2 until no variables remain.*
- The sequence of substitutions to obtain a string is called a ***derivation***. Derivation can be of two types:
  - In leftmost derivations, the leftmost nonterminal in each sentential is always chosen.
  - In rightmost derivations, the rightmost nonterminal is always chosen.

Derivations

- Consider the grammar:

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$$

- Leftmost derivation for –(id+id):

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id})$$

- Rightmost derivation for –(id+id):

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \mathbf{id}) \Rightarrow -(\mathbf{id} + \mathbf{id})$$

Parse Tree

- A parse tree is a graphical representation of a derivation.

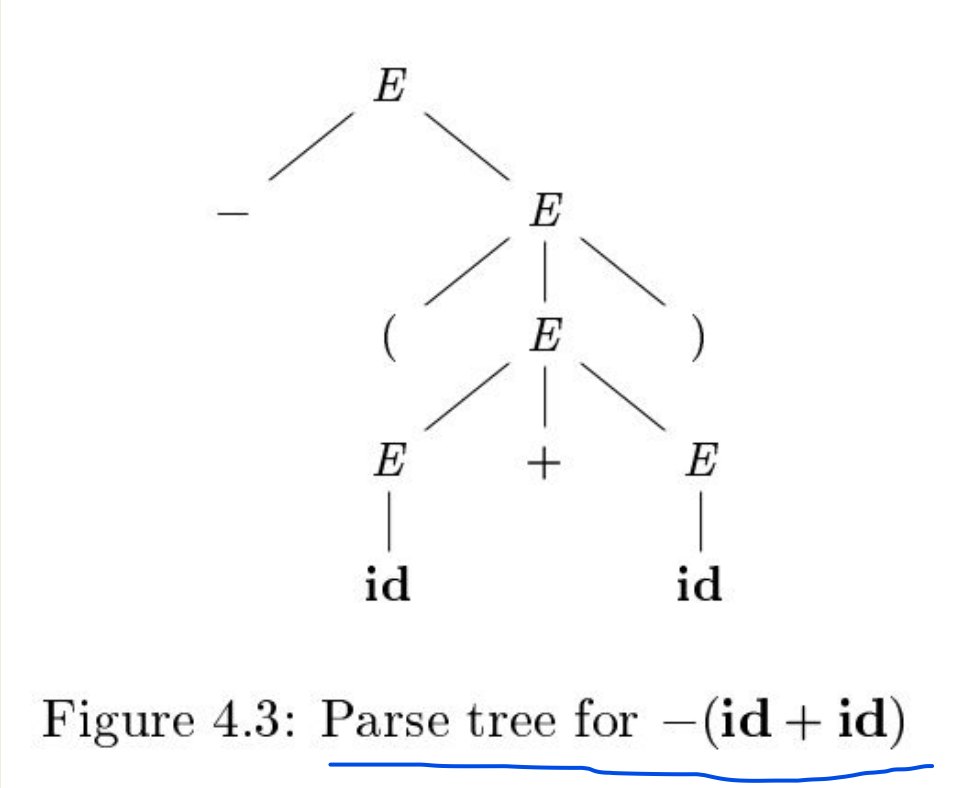
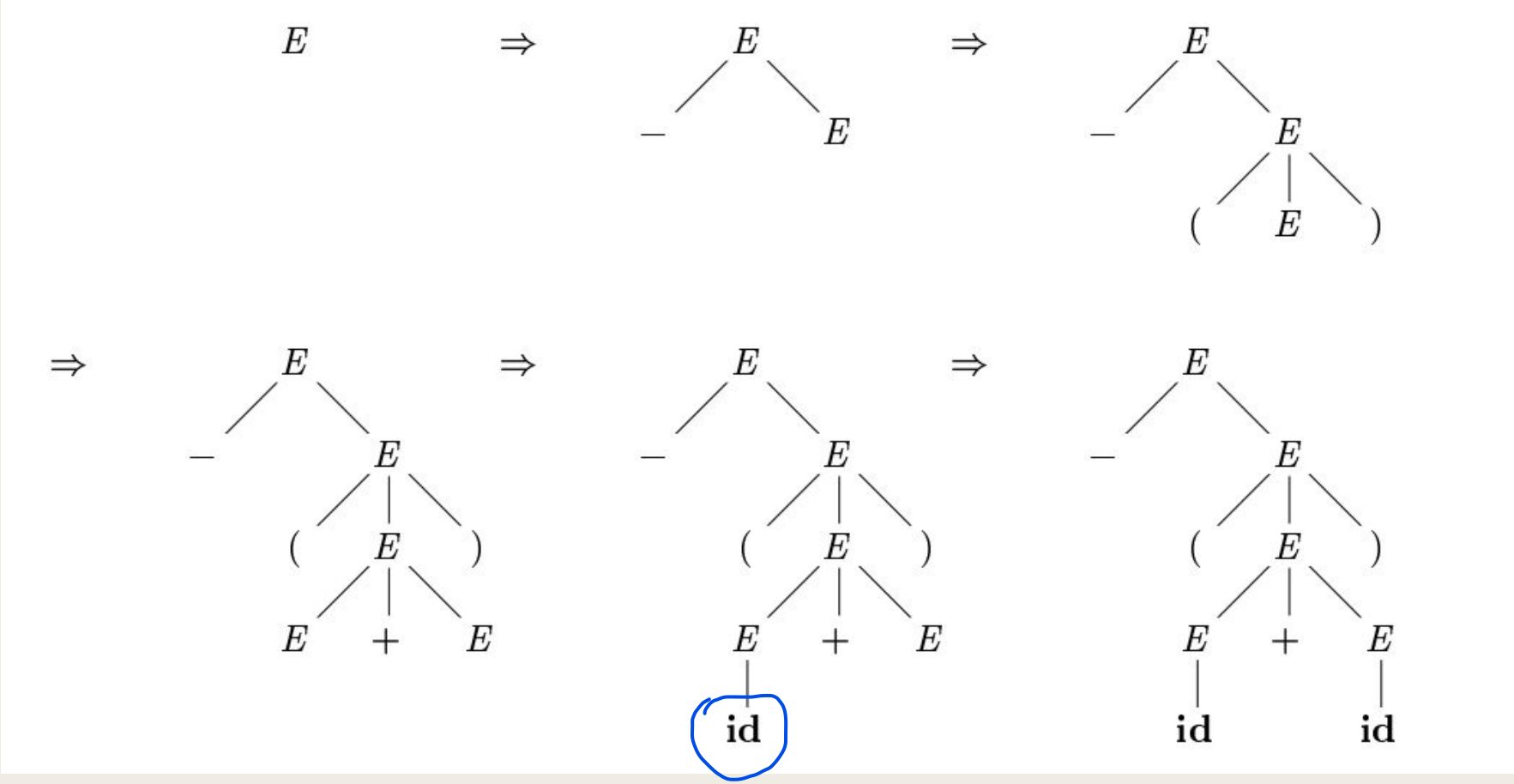


Figure 4.3: Parse tree for $-(\mathbf{id} + \mathbf{id})$

Parse Tree

Sequence of parse trees for leftmost derivation



Recursion in Grammar:

- If the leftmost symbol in the right hand side of a production rule is the same as the head of the production rule then, it is left recursive. That is, $A \rightarrow A\alpha$ is left recursive.
- If the rightmost symbol in the right hand side of a production rule is the same as the head of the production rule then, it is right recursive. That is, $A \rightarrow \alpha A$ is right recursive.

Ambiguity

- An ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.
- For grammars used to detect programming constructs we have to design unambiguous grammar so that the computer doesn't have to deal with a choice of which derivation to use.

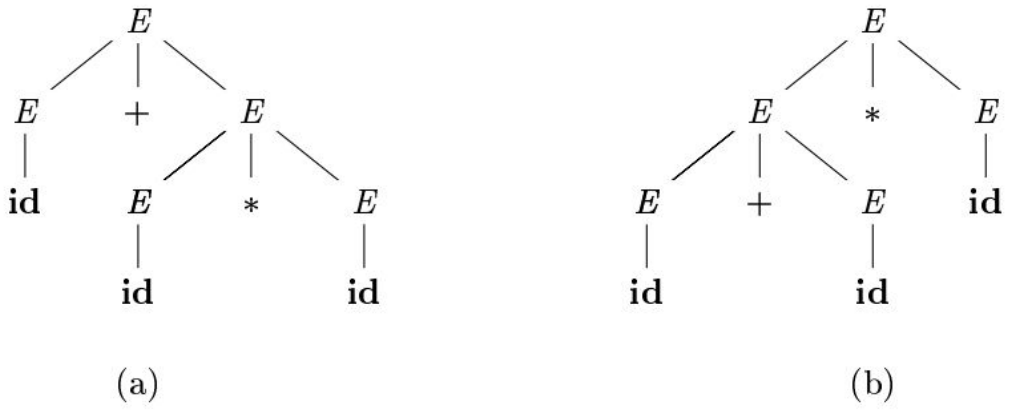


Figure 4.5: Two parse trees for **id+id*id**

Disambiguiting Rules

- Two main problems lead to ambiguity: problem in associativity and precedence.
- To handle associativity, we define the recursion of the grammar.
- To handle precedence, we define the grammar in levels.
- Let's try for these:

$$E \rightarrow E + E \mid E * E \mid E \hat{} E \mid id$$

$$bExp \rightarrow bExp \ \wedge \ bExp \mid bExp \ \vee \ bExp \mid \neg bExp \mid True \mid False$$

$$regex \rightarrow regex + regex \mid regex.regex \mid regex * \mid terminal$$

Understanding operator precedence and associativity

Let’s find the precedence and associativity of the operators for the following grammars:

$$A \rightarrow A \,\$\, B$$
$$B \rightarrow C : B$$
$$C \rightarrow C \,@\, D$$
$$D \rightarrow d$$

$$E \rightarrow E * F \mid F + E \mid F$$
$$F \rightarrow F - F \mid id$$

Eliminating Left Recursion

- A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.
- Top-down parsing methods (using leftmost derivation) cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.
- Immediate left recursion is when there is a production of the form $A \rightarrow A\alpha$.

Eliminating Left Recursion

- Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions.
- First, group the productions as

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where no β_i begins with an A. Then, replace the A-productions by

$$\begin{aligned} A &\to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' &\to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

The nonterminal A generates the same strings as before but is no longer left recursive.

Eliminating Left Recursion

- This procedure eliminates all left recursion from the A and A' productions (provided no α_i is ε), but it does not eliminate left recursion involving derivations of two or more steps.
- For example, consider the grammar:

$$S \rightarrow A\ a \mid b$$
$$A \rightarrow A\ c \mid S\ d \mid \epsilon$$

- The nonterminal S is left recursive because $S \Rightarrow A\ a \Rightarrow S\ d\ a$, but it is not immediately left recursive.

Eliminating Left Recursion

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i-1$) {
4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions
7) }

Eliminating Left Recursion

- Let's consider the following left recursive grammar:

$$A \rightarrow Bxy \mid x$$
$$B \rightarrow CD$$
$$C \rightarrow A \mid w$$
$$D \rightarrow z$$

Let $A_1 = A,\ A_2 = B,\ A_3 = C,\ A_4 = D.$

For i=1, the grammar remains unchanged.

For i=2, the grammar remains unchanged.

For i=3, j=1, we search for rules matching $A_3 \rightarrow A_1\gamma$ that is $C \rightarrow A\gamma$, we find $C \rightarrow A$. Now we replace the occurrence of A in the RHS of our matched rule with the RHS of A's production rules. The new grammar is:

$$A \rightarrow Bxy \mid x$$
$$B \rightarrow CD$$
$$C \rightarrow Bxy \mid x \mid w$$
$$D \rightarrow z$$

Eliminating Left Recursion

- *For i=3, j=2, we search for rules matching* $A_3 \rightarrow A_2\gamma$ *that is* $C \rightarrow B\gamma$*, we find* $C \rightarrow Bxy$*. Now we replace the occurrence of B in the RHS of our matched rule with the RHS of B's production rules. The new grammar is:*

$$A \rightarrow Bxy \mid x$$
$$B \rightarrow CD$$
$$C \rightarrow CDxy \mid x \mid w$$
$$D \rightarrow z$$

Before we move on to i=4, we have to remove the immediate left recursion. We detect $C \rightarrow CDxy$ *as immediate left recursion. So, we define a new variable* C' *and remove the left recursion to obtain:*

$$A \rightarrow Bxy \mid x$$
$$B \rightarrow CD$$
$$C \rightarrow xC' \mid wC'$$
$$C' \rightarrow DxyC' \mid \epsilon$$
$$D \rightarrow z$$

Eliminating Left Recursion

- *For i=4, the grammar remains unchanged.*

  So, our final grammar without left recursion is:

$$A \rightarrow Bxy \mid x$$
$$B \rightarrow CD$$
$$C \rightarrow xC' \mid wC'$$
$$C' \rightarrow DxyC'$$
$$D \rightarrow z$$

Left factoring

- 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. This is done in left factoring
- For example, if we have the two productions

$$stmt \rightarrow \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt$$
$$\mid \textbf{if } expr \textbf{ then } stmt$$

- When we see the input if, we cannot immediately tell which production to choose to expand stmt.

Method for Left factoring

- For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$, i.e., there is a nontrivial common prefix, we replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta \mid \ldots \mid \alpha\beta \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$A \to \alpha A' \mid \gamma$$
$$A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

- Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Method for Left factoring

- Let's left factor the if-else grammar from before:

$$stmt \rightarrow \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt$$
$$\mid \textbf{if } expr \textbf{ then } stmt$$

The longest common prefix is "$\textbf{if } expr \textbf{ then } stmt$", so by left factoring we have:

$$stmt \rightarrow \textbf{if } expr \textbf{ then } stmt\ S'$$
$$S' \rightarrow \textbf{else } stmt \mid \epsilon$$

Method for Left factoring

- If we consider the grammar:

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

$$E \rightarrow b$$

- We can see that $i\ E\ t\ S\ e\ i\ S$ is the longest prefix that is common between two productions.
- So our grammar becomes:

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

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

$$E \rightarrow b$$

- We still have common prefix, the longest common prefix now is $i\ E\ t\ S$.

Method for Left factoring

- Doing the same thing as before, our grammar becomes:

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

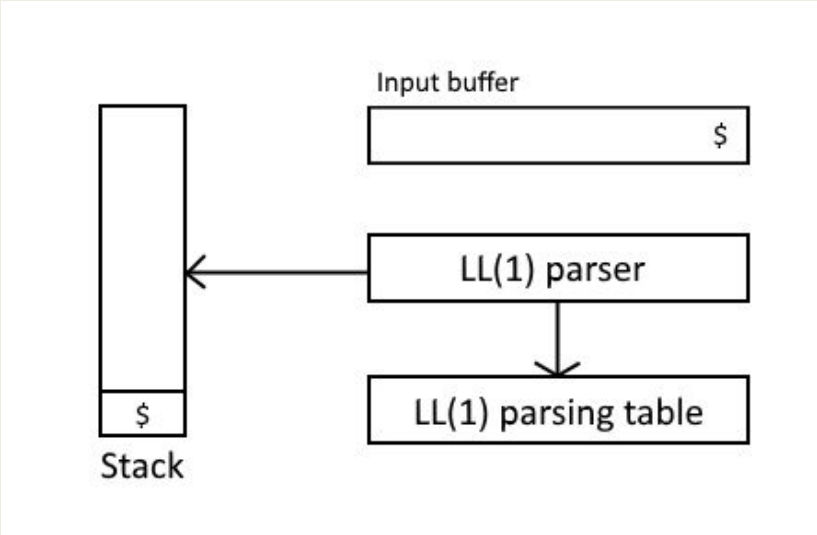
$$S'' \rightarrow e\ i\ S\ S' \mid \varepsilon$$

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

$$E \rightarrow b$$

Top-down parsing

- LL(1) parsing, which is a top-down parsing technique, works only for left factored unambiguous grammars without left recursion.
- The first ‘L’ in LL(1) means we are scanning the input from left to right. The second ‘L’ means we are using leftmost derivation and the ‘1’ means we have 1 input symbol of lookahead at each step to make parsing decisions.



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), where α is any string of grammar symbols, is defined as the set of terminals that begin strings derived from α. If $\alpha \overset{*}{\Rightarrow} \varepsilon$ then ε is also in FIRST(α).
- FOLLOW(A), for nonterminal A, is defined as the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form $S \overset{*}{\Rightarrow} \alpha A a \beta$, for some α and β.

Finding FIRST

- To compute FIRST(X) for all grammar symbols X , apply the following rules until no more terminals or ϵ can be added to any FIRST set.
  - If X is a terminal, then FIRST(X) = $\{X\}$
  - If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1),....., FIRST(Y_{i-1}). If ϵ is in FIRST(Y_j) for all j = 1, 2, ... , k, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \epsilon$, then we add FIRST(Y_2), and so on.
  - If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

Finding FOLLOW

- To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.
  - Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right end marker.
  - If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ in FOLLOW(B).
  - If there is a production $A \rightarrow \alpha B$, or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Finding FIRST and FOLLOW

Production Rules	FIRST	FOLLOW
$S \rightarrow ABCDE$	$\{a, b, c\}$	$\{\$\}$
$A \rightarrow a \mid \epsilon$	$\{a, \epsilon\}$	$\{b, c\}$
$B \rightarrow b \mid \epsilon$	$\{b, \epsilon\}$	$\{c\}$
$C \rightarrow c$	$\{c\}$	$\{d, e, \$\}$
$D \rightarrow d \mid \epsilon$	$\{d, \epsilon\}$	$\{e, \$\}$
$E \rightarrow e \mid \epsilon$	$\{e, \epsilon\}$	$\{\$\}$

Finding FIRST and FOLLOW

Production Rules	FIRST	FOLLOW
$S \rightarrow Bb \mid Cd$	$\{a, b, c, d\}$	$\{\$\}$
$B \rightarrow aB \mid \epsilon$	$\{a, \epsilon\}$	$\{b\}$
$C \rightarrow cC \mid \epsilon$	$\{c, \epsilon\}$	$\{d\}$

Finding FIRST and FOLLOW

Production Rules	FIRST	FOLLOW
$E \rightarrow TE'$	{**id**, (}	{$,)}
$E' \rightarrow +TE' \mid \epsilon$	{+, ϵ}	{$,)}
$T \rightarrow FT'$	{**id**, (}	{+, $,)}
$T' \rightarrow * FT' \mid \epsilon$	{ *, ϵ}	{+, $,)}
$F \rightarrow$ **id** $\mid (E)$	{**id**, (}	{*, +, $,)}

Finding FIRST and FOLLOW

Production Rules	FIRST	FOLLOW
$S \rightarrow ACB \mid CbB \mid Ba$	$\{d, g, h, \epsilon, b, a\}$	$\{\$\}$
$A \rightarrow da \mid BC$	$\{d, g, h, \epsilon\}$	$\{h, g, \$\}$
$B \rightarrow g \mid \epsilon$	$\{g, \epsilon\}$	$\{\$, a, h, g\}$
$C \rightarrow h \mid \epsilon$	$\{h, \epsilon\}$	$\{g, \$, b, h\}$

Finding FIRST and FOLLOW

Find the FIRST and FOLLOW for the following grammars:

$$S \rightarrow aBDh$$
$$B \rightarrow cC$$
$$C \rightarrow bC \mid \epsilon$$
$$D \rightarrow EF$$
$$E \rightarrow g \mid \epsilon$$
$$F \rightarrow f \mid \epsilon$$

$$E \rightarrow TE'$$
$$E' \rightarrow ?E \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow !T \mid \epsilon$$
$$F \rightarrow WF'$$
$$F' \rightarrow @F \mid \epsilon$$
$$W \rightarrow UV \mid V \mid [E]$$
$$V \rightarrow a \mid b$$
$$U \rightarrow <|>$$

Construction of LL(1) parsing table

Production Rules

Non-terminals	FIRST	FOLLOW
	{**id**, (}	{$,)}
	{+, ϵ}	{$,)}
	{**id**, (}	{+, $,)}
	{ *, ϵ}	{+, $,)}
	{**id**, (}	{*, +, $,)}

Construction of LL(1) parsing table

Production Rules

$FIRST(TE') = \{id, (\}$

Non-terminals	FIRST	FOLLOW
	{**id**, (}	{$,)}
	{+, ϵ}	{$,)}
	{**id**, (}	{+, $,)}
	{ $*$, ϵ}	{+, $,)}
	{**id**, (}	{$*$, +, $,)}

	$E \rightarrow TE'$			$E \rightarrow TE'$		

Construction of LL(1) parsing table

Production Rules

$FIRST(+TE') = \{+\}$

	$E \rightarrow TE'$			$E \rightarrow TE'$		
		$E' \rightarrow +TE'$				

Non-terminals	FIRST	FOLLOW
	{**id**, (}	{$,)}
	$\{+, \epsilon\}$	{$,)}
	{**id**, (}	$\{+, \$,)\}$
	$\{*, \epsilon\}$	$\{+, \$,)\}$
	{**id**, (}	$\{*, +, \$,)\}$

Construction of LL(1) parsing table

Production Rules

$FOLLOW(E') = \{\$,)\}$

	$E \rightarrow TE'$			$E \rightarrow TE'$		
		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$

Non-terminals	FIRST	FOLLOW
	{**id**, (}	{$,)}
	$\{+, \epsilon\}$	{$,)}
	{**id**, (}	$\{+, \$,)\}$
	$\{*, \epsilon\}$	$\{+, \$,)\}$
	{**id**, (}	$\{*, +, \$,)\}$

Construction of LL(1) parsing table

Production Rules

$FIRST(FT') = \{id, (\}$

Non-terminals	FIRST	FOLLOW
	{**id**, (}	{$,)}
	{+, ϵ}	{$,)}
	{**id**, (}	{+, $,)}
	{ *, ϵ}	{+, $,)}
	{**id**, (}	{*, +, $,)}

	$E \rightarrow TE'$			$E \rightarrow TE'$		
		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
	$T \rightarrow FT'$			$T \rightarrow FT'$		

Construction of LL(1) parsing table

Production Rules

$FIRST(*\ FT') = \{*\}$

Non-terminals	FIRST	FOLLOW
	{**id**, (}	{$,)}
	{+, ϵ}	{$,)}
	{**id**, (}	{+, $,)}
	{ *, ϵ}	{+, $,)}
	{**id**, (}	{*, +, $,)}

	$E \rightarrow TE'$			$E \rightarrow TE'$		
		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
	$T \rightarrow FT'$			$T \rightarrow FT'$		
			$T' \rightarrow * FT'$			

Construction of LL(1) parsing table

Production Rules

$FOLLOW(T') = \{+, \$,)\}$

Non-terminals	FIRST	FOLLOW
	{**id**, (}	{$,)}
	{+, ϵ}	{$,)}
	{**id**, (}	{+, $,)}
	{ $*$, ϵ}	{+, $,)}
	{**id**, (}	{$*$, +, $,)}

	$E \rightarrow TE'$			$E \rightarrow TE'$		
		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
	$T \rightarrow FT'$			$T \rightarrow FT'$		
		$T' \rightarrow \epsilon$	$T' \rightarrow * FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$

Construction of LL(1) parsing table

Production Rules

$FIRST((E)) = \{(\}$

Non-terminals	FIRST	FOLLOW
	{**id**, (}	{$,)}
	{+, ϵ}	{$,)}
	{**id**, (}	{+, $,)}
	{*, ϵ}	{+, $,)}
	{**id**, (}	{*, +, $,)}

	$E \rightarrow TE'$			$E \rightarrow TE'$		
		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
	$T \rightarrow FT'$			$T \rightarrow FT'$		
		$T' \rightarrow \epsilon$	$T' \rightarrow * FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
				$F \rightarrow (E)$		

Construction of LL(1) parsing table

Production Rules

$FIRST(\mathbf{id}) = \{\mathbf{id}\}$

Non-terminals	FIRST	FOLLOW
	$\{\mathbf{id}, (\}$	$\{\$,)\}$
	$\{+, \epsilon\}$	$\{\$,)\}$
	$\{\mathbf{id}, (\}$	$\{+, \$,)\}$
	$\{*, \epsilon\}$	$\{+, \$,)\}$
	$\{\mathbf{id}, (\}$	$\{*, +, \$,)\}$

	$E \to TE'$			$E \to TE'$		
		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
	$T \to FT'$			$T \to FT'$		
		$T' \to \epsilon$	$T' \to * FT'$		$T' \to \epsilon$	$T' \to \epsilon$
	$F \to \mathbf{id}$			$F \to (E)$		

LL(1) predictive parsing

- Let's first do it with a simple grammar

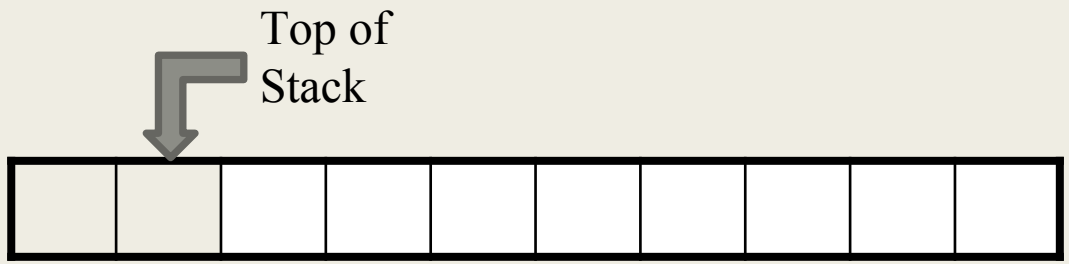
$$S \rightarrow (S) \mid \epsilon$$

Production Rules

Non-terminals	FIRST	FOLLOW

LL(1) predictive parsing

Input String: (( )) $

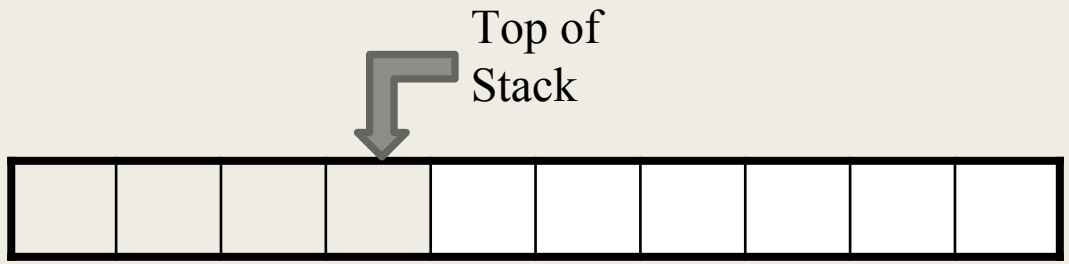


Parse Tree:

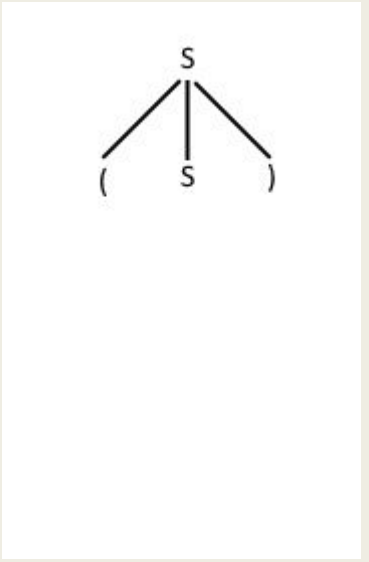
Top of stack gives nonterminal S and we get (from the input pointer, so we push (S) into the stack

LL(1) predictive parsing

Input String: (( )) $



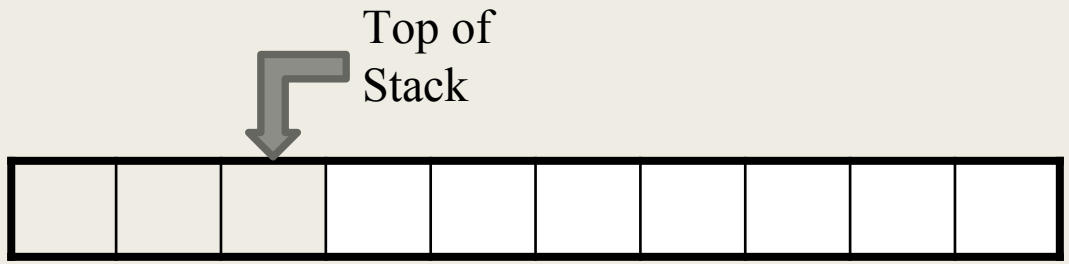
Parse Tree:



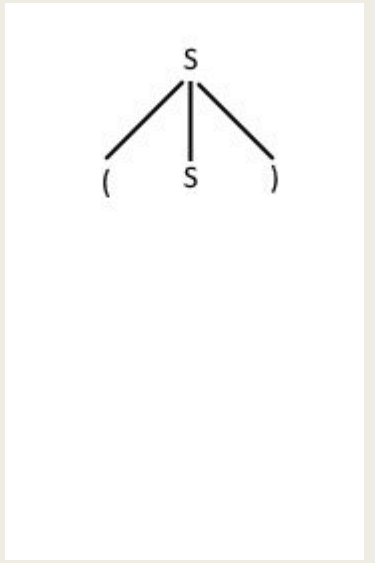
When we push a RHS of a production rule in to the stack, it means we have used that rule to expand our tree. The terminal at the top of stack matches with the terminal in input so we pop it off and move the input pointer forward.

LL(1) predictive parsing

Input String: (( )) $



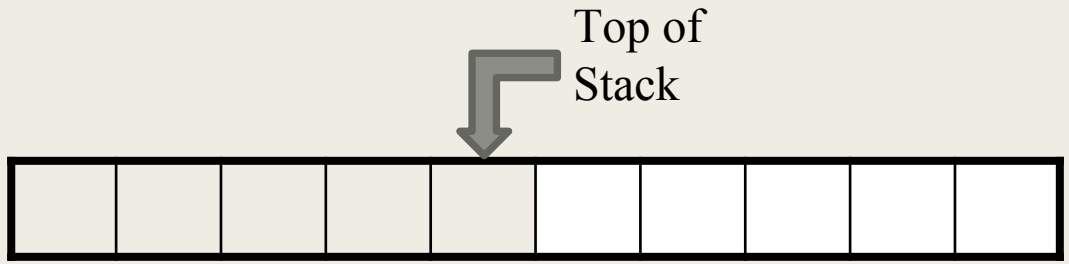
Parse Tree:



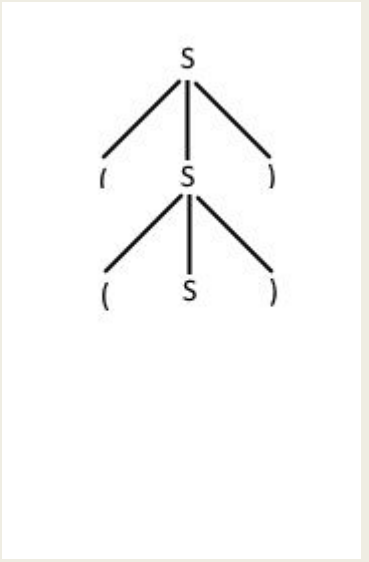
Top of stack gives nonterminal *S* and we get (from the input pointer , so we push (*S*) into the stack

LL(1) predictive parsing

Input String: (( )) $



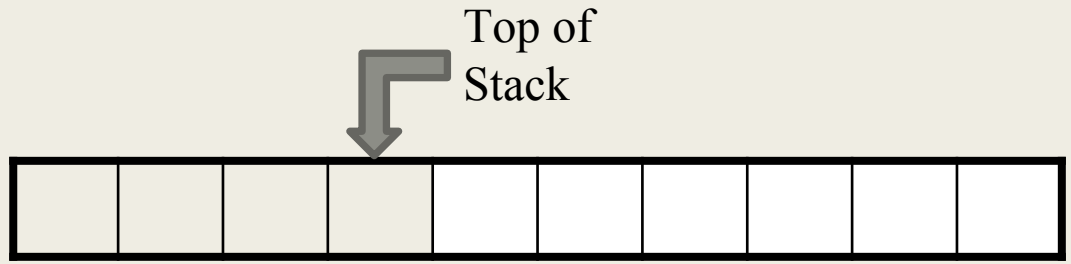
Parse Tree:



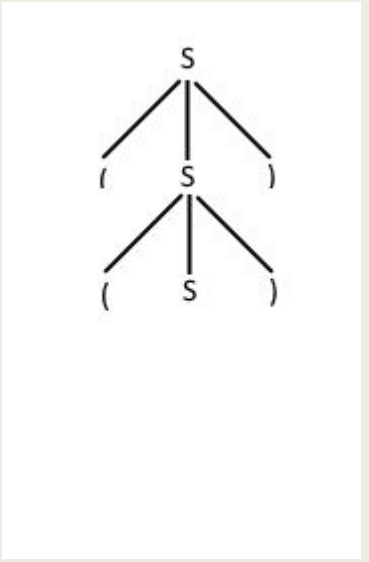
The terminal at the top of stack matches with the terminal in input so we pop it off and move the input pointer forward.

LL(1) predictive parsing

Input String: (( )) $



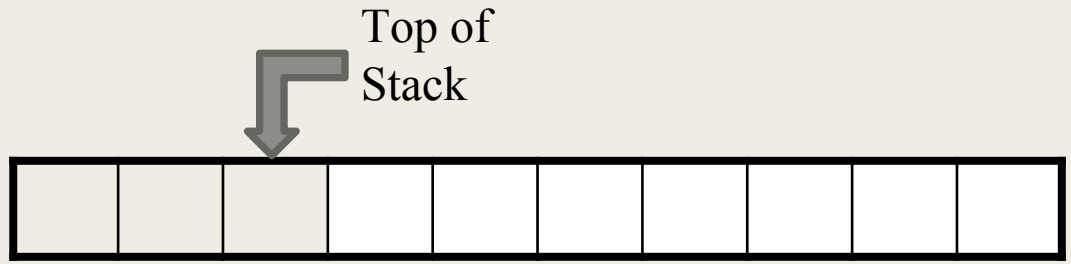
Parse Tree:



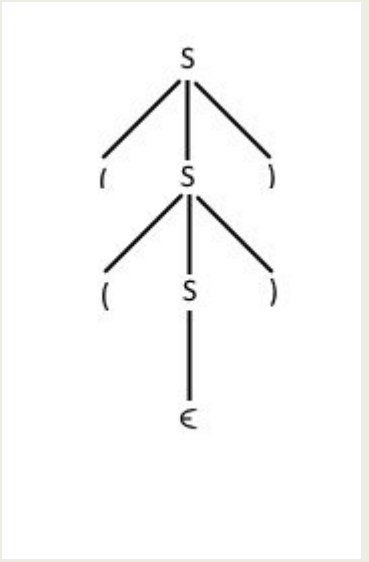
Top of stack gives nonterminal S and we get (from the input pointer, the matched rule is $S \rightarrow \epsilon$ so we simply pop S.

LL(1) predictive parsing

Input String: (( )) $



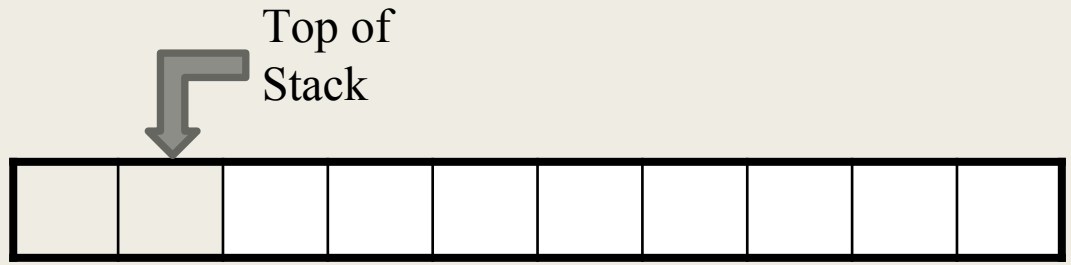
Parse Tree:



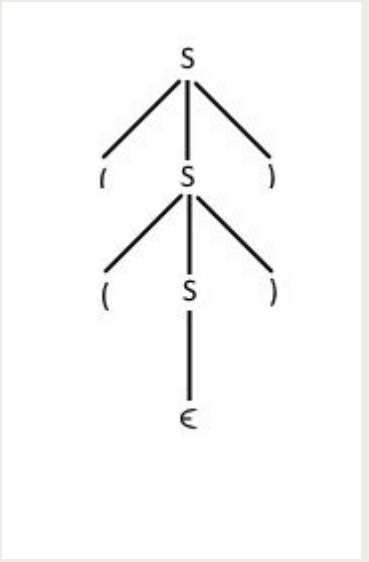
The terminal at the top of stack matches with the terminal in input so we pop it off and move the input pointer forward.

LL(1) predictive parsing

Input String: (( )) $



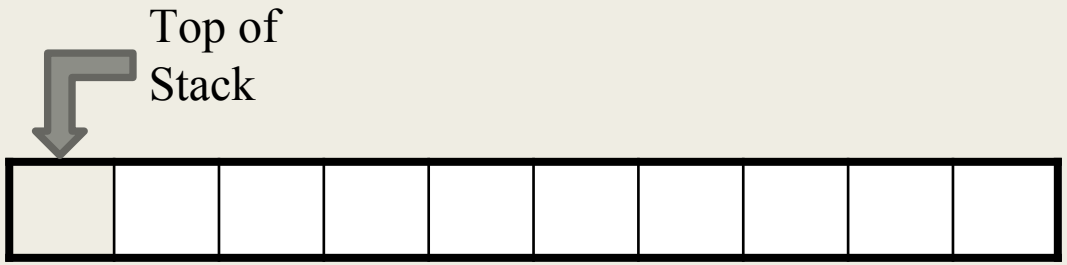
Parse Tree:



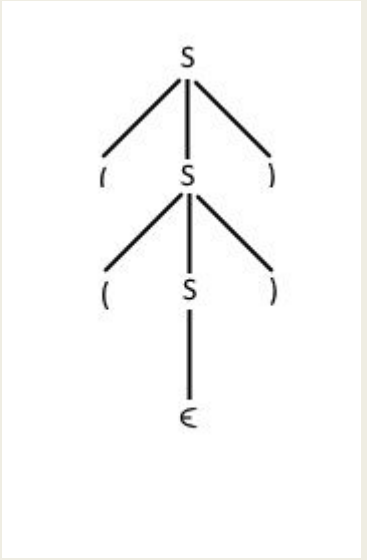
The terminal at the top of stack matches with the terminal in input so we pop it off and move the input pointer forward.

LL(1) predictive parsing

Input String: (( )) $



Parse Tree:



As the $ symbol has been matched, our parsing is complete.

LL(1) Predictive Parsing

- Any grammar that causes two production rules to be entered into the same slot of the parsing table cannot be parsed with LL(1) predictive parsing.
- Grammars that are ambiguous or not left factored or left recursive will put more than one rule into the same slot.
- For productions $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$, if the FIRST($\alpha_1$), FIRST($\alpha_2$), …, FIRST($\alpha_{n-1}$) and FIRST($\alpha_n$) are not all mutually exclusive then we will have more than one rule put into the same slot.
- If we have a production rule such as $A \rightarrow \alpha \mid \epsilon$ and FIRST(α) shares element with FOLLOW(A) then both $A \rightarrow \alpha$ and $A \rightarrow \epsilon$ will be put into the same slot.

String Validation

For the Input : id+id*id

For the Grammar

- E -> T E
- 'E' -> + T E' | Ɛ
- T -> F T‘
- T' -> * F T' | Ɛ
- F -> id

String Validation

Stack	Input	Moves
E$	Id+id*id $	E-> TE’
TE’$	Id+id*id $	T->FT’
FT’E’$	Id+id*id $	F->id
idT’E’$	Id+id*id $	
T’E’$	+id*id $	T’-> Ɛ
E’$	+id*id $	E’->+TE’
+TE’$	+id*id $	
TE’$	id*id $	T->FT’
FT’E’$	id*id $	F->id
idT’E’$	id*id $	
T’E’$	*id $	T’->*FT’
*FT’E’$	*id $	
FT’E’$	id $	F->id
idT’E’$	id $	
T’E’$	$	T’-> Ɛ
E’$	$	E’-> Ɛ
$	$	ACCEPT

	id	+	*	$
E	$E \rightarrow TE'$			
E'		$E' \rightarrow +TE'$		$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			
T'		$T' \rightarrow \epsilon$	$T' \rightarrow * FT'$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			

For the Input : id+id*id

For the Grammar

E -> T E’

E' -> + T E' | Ɛ

T -> F T‘

T' -> * F T' | Ɛ

F -> id