S J P N Trust's

HIRASUGAR INSTITUTE OF TECHNOLOGY, NIDASOSHI.

Inculcating Values, Promoting Prosperity

Approved by AICTE, Recognized by Govt. of Karnataka and Permanently Affiliated to VTU Belagavi.

Accredited at 'A' Grade by NAAC

Programmes Accredited by NBA: CSE, ECE, EEE & ME

Subject: System Software and Compilers (18CS61)

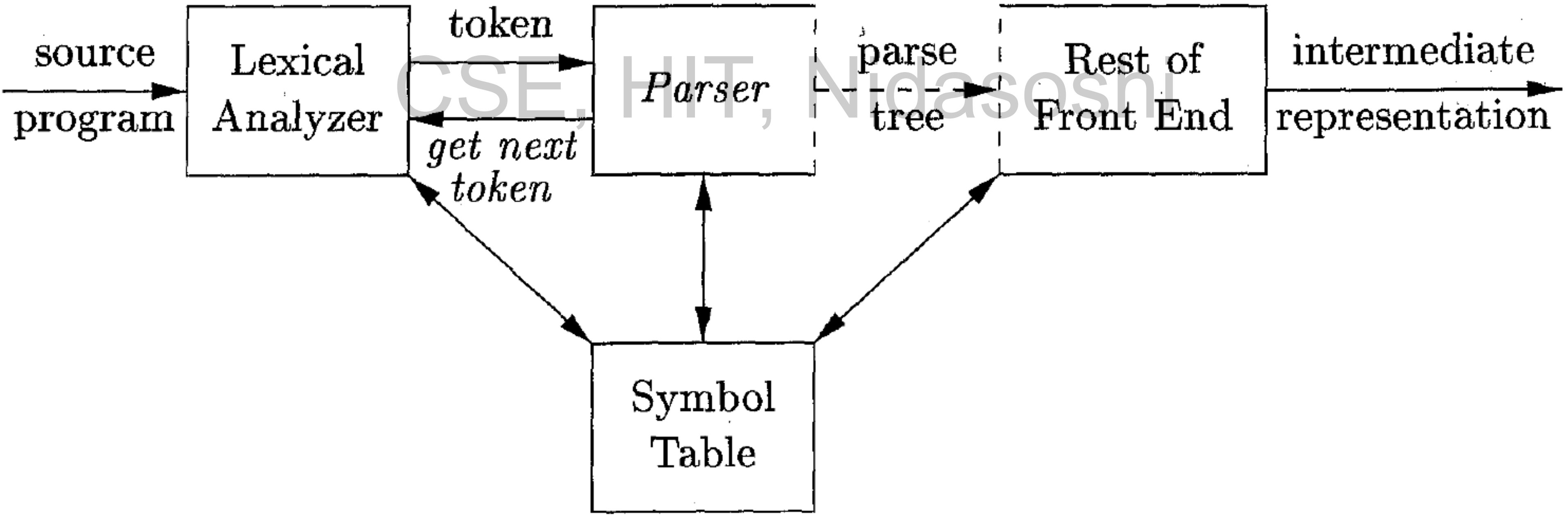
Module 3: Syntax Analysis

Dr. Mahesh G. Huddar

Dept. of Computer Science and Engineering

The Role of the Parser

- In compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in Fig, and verifies that the string of token names can be generated by the grammar for the source language.



The Role of the Parser

- We expect 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.
- Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.
- There are **three** general types of parsers for grammars: **universal**, **top-down**, and **bottom-up**.
- Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar

The Role of the Parser

- The methods commonly used in compilers can be classified as being either **top-down or bottom-up**.
- As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves).
- Bottom-up methods start from the leaves and work their way up to the root.
- In either case, the input to the parser is scanned from left to right, one symbol at a time.

Syntax 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.
- Strikingly, few languages have been designed with error handling in mind, even though errors are so commonplace.
- Most programming language specifications do not describe how a compiler should respond to errors; error handling is left to the compiler designer.
- Planning the error handling right from the start can both simplify the structure of a compiler and improve its handling of errors.

Syntax Error Handling

Common programming errors can occur at many different levels.

1. ***Lexical errors*** include misspellings of identifiers, keywords, or operators - e.g., the use of an identifier elipsesize instead of ellipsesize – and missing quotes around text intended as a string.
2. ***Syntactic errors*** include misplaced semicolons or extra or missing braces.
3. ***Semantic errors*** include type mismatches between operators and operands. An example is a return statement in a Java method with result type void.
4. ***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 ==.

Syntax 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.

Error-Recovery Strategies

- Once an error is detected, how should the parser recover? Although no strategy has proven itself universally acceptable, a few methods have broad applicability.
- The simplest approach is for the parser to quit with an informative error message when it detects the first error.
- The following recovery strategies implemented in parser: **panic-mode, phrase-level, error-productions, and global-correction.**

Error-Recovery Strategies

1. 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.
- The compiler designer must select the synchronizing tokens appropriate for the source language.
- While panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has the advantage of simplicity, and, unlike some methods to be considered later, is guaranteed not to go into an infinite loop.
- **Example: int a, 5abcd, sum, $2;**

Error-Recovery Strategies

Advantage:

1. It’s easy to use.
2. The program never falls into the loop.

Disadvantage:

1. This technique may lead to runtime error in further stages.

Error-Recovery Strategies

2. 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

- **Example:**
- **int a,b**
- **// AFTER RECOVERY:**
- **int a,b;**

- **Advantages:** This method is used in many errors repairing compilers.
- **Disadvantages:** While doing the replacement the program should be prevented from falling into an infinite loop.

Error-Recovery Strategies

3. 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 Strategies

Example: Suppose the input string is **abcd**.

Grammar: S-> A

A-> aA | bA | a | b

B-> cd

Grammar: E->SB // AUGMENT THE GRAMMAR

S-> A

A-> aA| bA | a | b

B-> cd

Now, string **abcd** is possible to obtain.

Error-Recovery Strategies

4. 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

- Grammars systematically describe the syntax of programming language constructs like expressions and statements.

- Using a syntactic variable ***stmt*** to denote statements and variable ***expr*** to denote **expressions**, the production

$$stmt \rightarrow \textbf{if} \ (\ expr \ ) \ stmt \ \textbf{else} \ stmt$$

- specifies the structure of **conditional** statement.

The Formal Definition of a Context-Free Grammar

A context-free grammar (grammar for short) consists of terminals, non-terminals, a start symbol, and productions.

1. ***Terminals*** are the basic symbols from which strings are formed. In the previous example, the terminals are the keywords **if** and **else** and the symbols "(" and ")".
2. ***Non-terminals*** are syntactic variables that denote sets of strings. In the previous example, ***stmt*** and ***expr*** are non-terminals.

The Formal Definition of a Context-Free Grammar

3. In a grammar, one nonterminal is distinguished as the **start symbol**. Conventionally, the productions for the start symbol are listed first.
4. The **productions** of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings.

   Each production consists of:

   a) A **nonterminal** called the **head or left side** of the production.

   b) The symbol **→**. or **::=**.

   c) A body or right side consisting of zero or more terminals and non-terminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

The Formal Definition of a Context-Free Grammar

Example:

- The below grammar defines simple arithmetic expressions.
- In this grammar, the terminal symbols are **id, +, -, *, /, (,)** and The nonterminal symbols are ***expression***, ***term*** and ***factor***, and ***expression*** is the start symbol

$$
\begin{aligned}
expression &\rightarrow expression + term \\
expression &\rightarrow expression - term \\
expression &\rightarrow term \\
term &\rightarrow term * factor \\
term &\rightarrow term \,/\, factor \\
term &\rightarrow factor \\
factor &\rightarrow (\ expression\ ) \\
factor &\rightarrow \mathbf{id}
\end{aligned}
$$

The Formal Definition of a Context-Free Grammar

Notational Conventions:

1. The symbols are terminals:

a) Lowercase letters in the alphabet, such as **a, b, c.**

b) Operator symbols such as **+, *,** and so on.

c) Punctuation symbols such as parentheses, comma, and so on.

d) The digits **0,1,. . . ,9**.

e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.

The Formal Definition of a Context-Free Grammar

2. The symbols are non-terminals:

a) Uppercase letters early in the alphabet, such as **A, B, C.**

b) The letter **S**, which, when it appears, is usually the start symbol.

c) Lowercase, italic names such as ***expr*** or ***stmt***.

d) When discussing programming constructs, uppercase letters may be used to represent non-terminals for the constructs. For example, non-terminals for **expressions**, **terms**, and **factors** are often represented by **E, T, and F**, respectively.

The Formal Definition of a Context-Free Grammar

3. A set of productions $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, . . . , $A \rightarrow \alpha_k$ with a common head A (call them A-productions), may be written $A \rightarrow \alpha_1 \mid \alpha_2 \cdots\cdots \mid \alpha_k$. Call $\alpha_1, \alpha_2, \cdots\cdots \alpha_k$ the alternatives for A.

4. Unless stated otherwise, the head of the first production is the start symbol.

The Formal Definition of a Context-Free Grammar

- Using these conventions, the previous grammar can be rewritten concisely as

$$
\begin{array}{lcl}
expression & \rightarrow & expression + term \\
expression & \rightarrow & expression - term \\
expression & \rightarrow & term \\
term & \rightarrow & term * factor \\
term & \rightarrow & term \, / \, factor \\
term & \rightarrow & factor \\
factor & \rightarrow & (\, expression \,) \\
factor & \rightarrow & \mathbf{id}
\end{array}
$$

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T \, / \, F \mid F \\
F & \rightarrow & (\, E \,) \mid \mathbf{id}
\end{array}
$$

Context-Free Grammar - Derivations

- The construction of a parse tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules.
- Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions.
- This derivational view corresponds to the top-down construction of a parse tree

Context-Free Grammar - Derivations

- For example, consider the following grammar,

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

- The production E → - E signifies that if E denotes an expression, then – E must also denote an expression.
- The replacement of a single E by - E will be described by writing

$$E \Rightarrow -E$$

- which is read, "E derives -E."

Context-Free Grammar - Derivations

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

- The production E → (E) can be applied to replace any instance of E in any string of grammar symbols by (E).
- We can take a single E and repeatedly apply productions in any order to get a sequence of replacements.

- For example,

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

- We call such a sequence of replacements a ***derivation*** of -(id) from E.

Context-Free Grammar - Derivations

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

- The string -(id + id) can be derived as shown below,

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

- Alternate derivation,

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

Context-Free Grammar - Derivations

To understand how parsers work, we shall consider derivations in which the nonterminal to be replaced at each step is chosen as follows:

1. In *lefimost* derivations, the leftmost nonterminal in each sentential is always chosen.

$$\alpha \underset{lm}{\Rightarrow} \beta.$$

2. In *rightmost* derivations, the rightmost nonterminal is always chosen

$$\alpha \underset{rm}{\Rightarrow} \beta$$

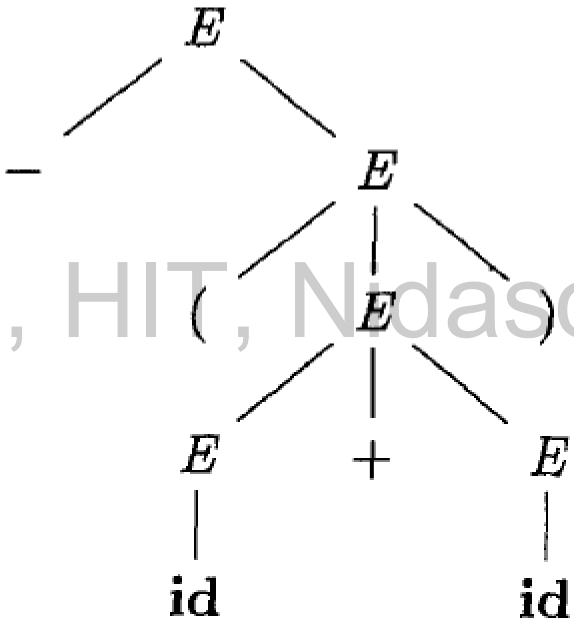
$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$

Context-Free Grammar - Parse Trees and Derivations

- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals.
- Each interior node of a parse tree represents the application of a production.

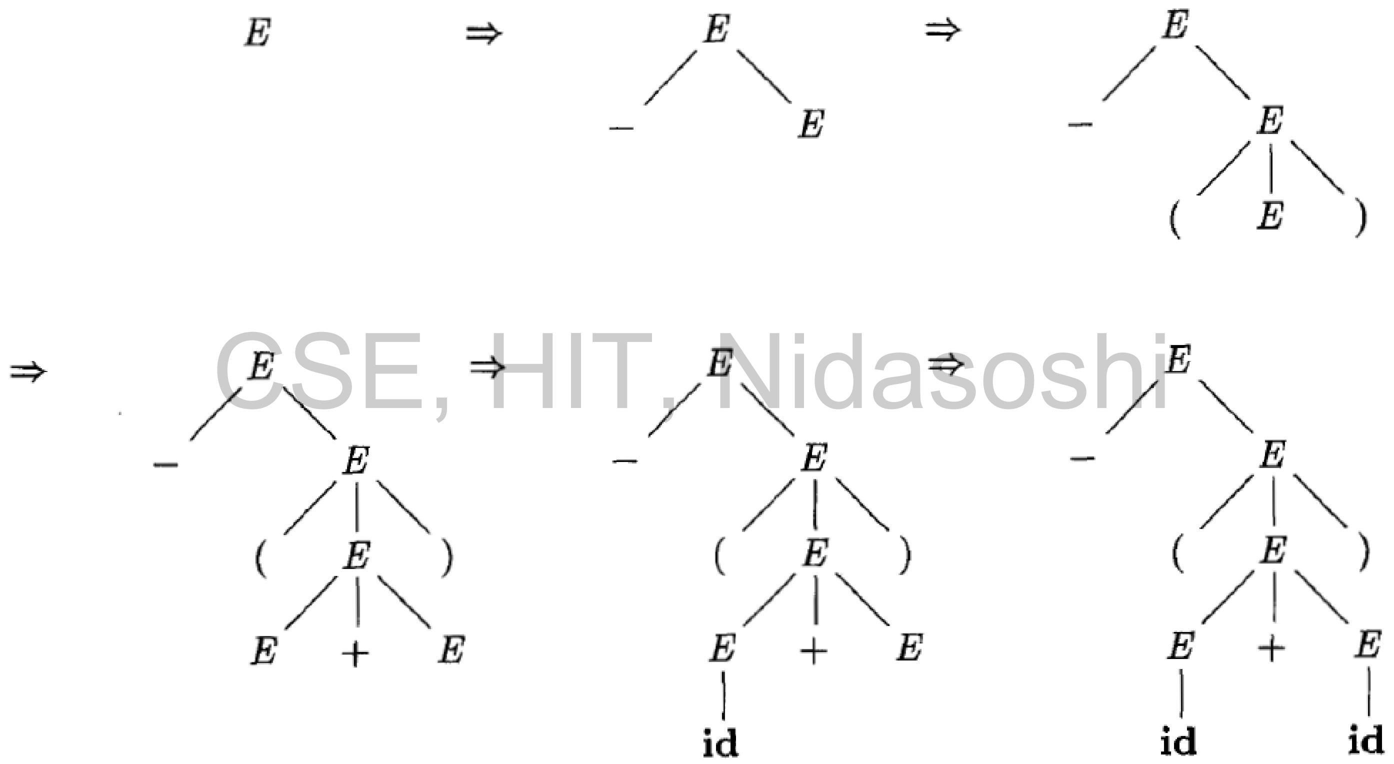
- The interior node is labeled with the ont terminal A in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.

Context-Free Grammar - Parse Trees and Derivations



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

Context-Free Grammar - Parse Trees and Derivations



Sequence of parse trees for derivation

Context-Free Grammar - Ambiguity

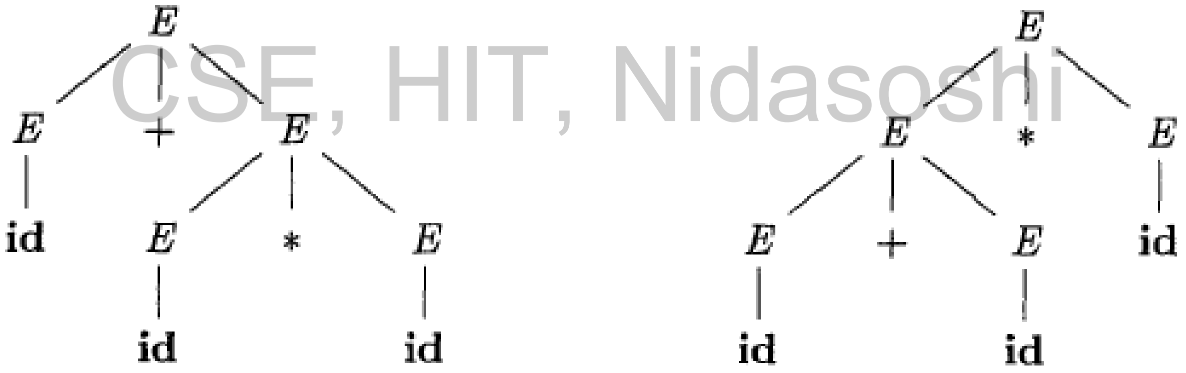
- A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*.
- Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.
- For example: Derivation for → **id + id * id** with below gramer

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

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \textbf{id} + E \\ &\Rightarrow \textbf{id} + E * E \\ &\Rightarrow \textbf{id} + \textbf{id} * E \\ &\Rightarrow \textbf{id} + \textbf{id} * \textbf{id} \end{aligned} \qquad \begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow \textbf{id} + E * E \\ &\Rightarrow \textbf{id} + \textbf{id} * E \\ &\Rightarrow \textbf{id} + \textbf{id} * \textbf{id} \end{aligned}$$

Context-Free Grammar - Ambiguity

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow \mathbf{id} + E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$



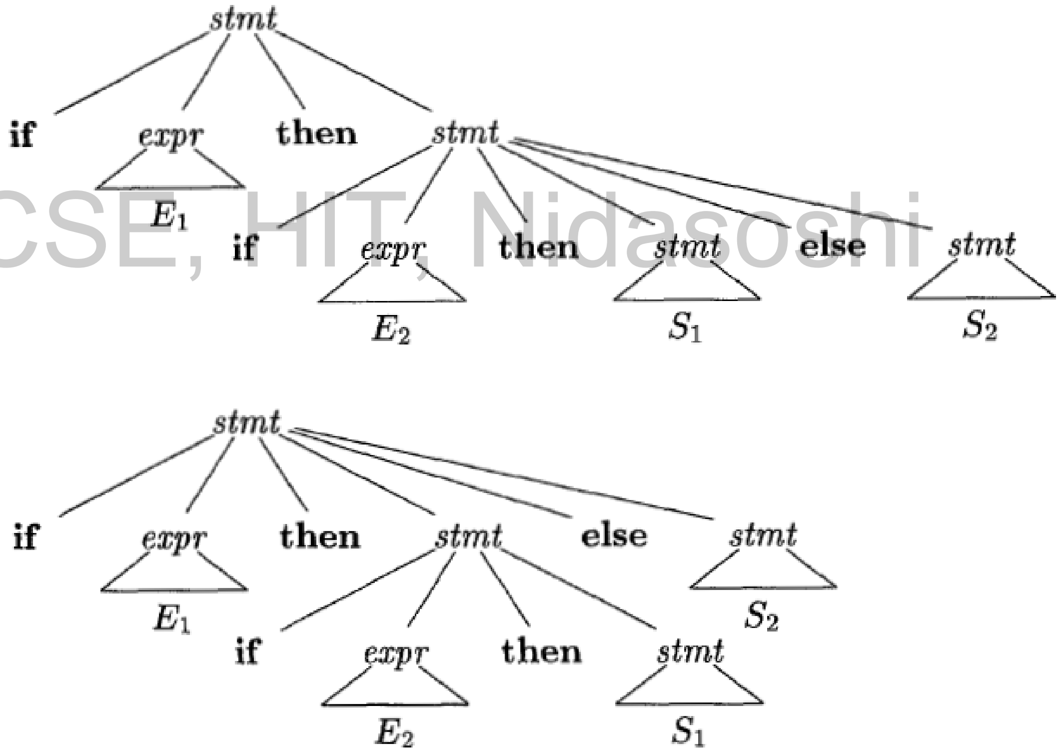
(a) (b)

Two parse trees for **id+id*id**

Context-Free Grammar - Ambiguity

$$
\begin{aligned}
stmt \;\rightarrow\; & \textbf{if } expr \textbf{ then } stmt \\
\mid\; & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
\mid\; & \textbf{other}
\end{aligned}
$$

if E_1 **then if** E_2 **then** S_1 **else** S_2



Context-Free Grammar - Ambiguity

- In all programming languages with conditional statements of this form, the first parse tree is preferred.
- The general rule is, “Match each else with the closest unmatched then.”

- This disambiguating rule can theoretically be incorporated directly into a grammar, but in practice it is rarely built into the productions.

Context-Free Grammar - Ambiguity

$$
\begin{array}{lll}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

- **Unambiguous grammar**

$$
\begin{array}{lll}
stmt & \rightarrow & matched\_stmt \\
 & | & open\_stmt \\
matched\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } matched\_stmt \\
 & | & \textbf{other} \\
open\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } open\_stmt
\end{array}
$$

Elimination of Left Recursion

- *A* grammar is *left recursive* if it has a nonterminal A such that there is a derivation

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

- Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

Elimination of Left Recursion

- **Immediate Left Recursion**

A Grammar is said to be left recursive grammars, if the first symbol in the right hand side of the production is same as the left hand side variable

Example:

E → E+T | T

T → T * | F

F → (E) | id

In this grammar, the first two productions

E → E+T

T → T * F

Elimination of Left Recursion

- **Indirect Left Recursion**

A Grammar is said to be Indirect left recursive grammar, if the first symbol in the right hand side of any of its derivations is same as the left hand side variable

Example:

$E \rightarrow T$

$T \rightarrow F$

$F \rightarrow E+T \mid id$

Here the indirect derivation is

$$E \Rightarrow T \Rightarrow F \Rightarrow E+T$$

The partial derivation E + T contains the first symbol E same as the LHS.

Elimination of Left Recursion

- A left-recursive pair of productions

$$A \to A\alpha \mid \beta$$

- could be replaced by the non-left-recursive productions:

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \epsilon$$

- without changing the strings derivable from A. This rule by itself suffices for many grammars

Elimination of Left Recursion

Elimination of Left Recursion

The left recursive grammar is of the form

$$A \Rightarrow A\alpha_1 \mid A\alpha_2 \ldots \mid \beta_1 \mid \beta_2 \ldots\ldots$$

Steps to replace left recursive productions as non-left recursive productions

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \ldots \mid \varepsilon$$

Elimination of Left Recursion

$$S \rightarrow Sa \mid Sb \mid c \mid d$$

$$A\alpha_1 \quad A\alpha_2 \quad \beta_1 \quad \beta_2$$

$$S \rightarrow cS' \mid dS'$$

$$S' \rightarrow aS' \mid bS' \mid \varepsilon$$

Elimination of Left Recursion

- **Eliminate Immediate Recursion**

E → E+T | T

T → T * F | F

F → (E) | id

Elimination of Left Recursion

- **Eliminate Immediate Recursion**

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

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

① $E \rightarrow E+T \mid T$ (E = A, E = A, +T = α, T = β)

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

② $T \rightarrow T*F \mid F$ (T = A, T = A, *F = α, F = β)

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

Final Grammar

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

Elimination of Left Recursion

- **Eliminate Indirect Recursion**

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \varepsilon$$

Elimination of Left Recursion

- **Eliminate Indirect Recursion**

$S \rightarrow Aa \mid b$

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

$S \rightarrow Aa \rightarrow Sda$

$A \rightarrow A\alpha_1 \mid A\alpha_2 \ldots \mid \beta_1 \mid \beta_2 \ldots$

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

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \varepsilon$

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

$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

(A, $A\ \alpha_1$, $A\ \alpha_2$, β_1, β_2)

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

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

Final Grammar

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

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

Elimination of Left Recursion

- **Eliminate Indirect Recursion**

$$S \rightarrow (L) \,/\, a$$

$$L \rightarrow L\, ,\, S \,/\, S$$

Elimination of Left Recursion

- **Eliminate Indirect Recursion**

$S \rightarrow (L) / a$

$L \rightarrow L , S / S$

Final Grammar

$S \rightarrow (L) / a$

$L \rightarrow SL'$

$L' \rightarrow , SL' / \varepsilon$

$L \rightarrow L , S / S$

$A \quad A \; \alpha \quad \beta$

$L \rightarrow SL'$

$L' \rightarrow , SL' / \varepsilon$

$A \rightarrow A\alpha / \beta$

$\Downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \varepsilon$

Left Factoring

- Left factoring is a grammar transformation helps to produce suitable grammar for Predictive parsing
- It is the process of converting non-deterministic grammar to Deterministic Grammar

- **Basic Idea:** When two alternative productions are available to expand a particular non-terminal with same prefix, there is a confusion to choose which production to apply for a non-terminal A.
- So rewrite the A-productions in-order to make right choice

Left Factoring

Consider there are two A-productions

$$A \rightarrow \alpha\beta_1|\alpha\beta_2.......| r_1 | r_2......$$

Which production to apply for the non-terminal A, $\alpha\beta_1$ or $\alpha\beta_2$

Convert the above production as

$$A \rightarrow \alpha A' | r_1 | r_2$$

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

Left Factoring

- **Example 1:**

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

Left Factoring

- $S \rightarrow iEtS\epsilon \mid iEtSeS \mid a$
  (α β_1 α β_2 γ_1)

  $E \rightarrow b$

  $S \rightarrow iEtSS' \mid a$

  $S' \rightarrow \epsilon \mid eS$

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \ldots\ldots \mid r_1 \mid r_2 \ldots\ldots$

⇩

$A \rightarrow \alpha A' \mid r_1 \mid r_2$

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

Final Grammar

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow \epsilon \mid eS$

$E \rightarrow b$

Left Factoring

- **Example 2:**

$$S \rightarrow a \ / \ abSb \ / \ aA$$

$$A \rightarrow bS \ / \ aAAb$$

Left Factoring

- **Example 2:**

$S \rightarrow a \,/\, abSb \,/\, aA$

$\rightarrow bS \,/\, aAAb$

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 \ldots\ldots | r_1 | r_2 \ldots\ldots$

⇩

$A \rightarrow \alpha A' | r_1 | r_2$

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

$S \rightarrow a\epsilon \,|\, abSb \,|\, aA$

$\alpha\beta_1 \quad \alpha\beta_2 \quad \alpha\beta_3$

$S \rightarrow aS'$

$S' \rightarrow \epsilon \,|\, bSb \,|\, A$

Final Grammar

$S \rightarrow aS'$

$S' \rightarrow \epsilon \,|\, bSb \,|\, A$

$A \rightarrow bS \,|\, aAAb$

CSE, HIT, Nidasoshi

Top-Down Parsing

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

- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

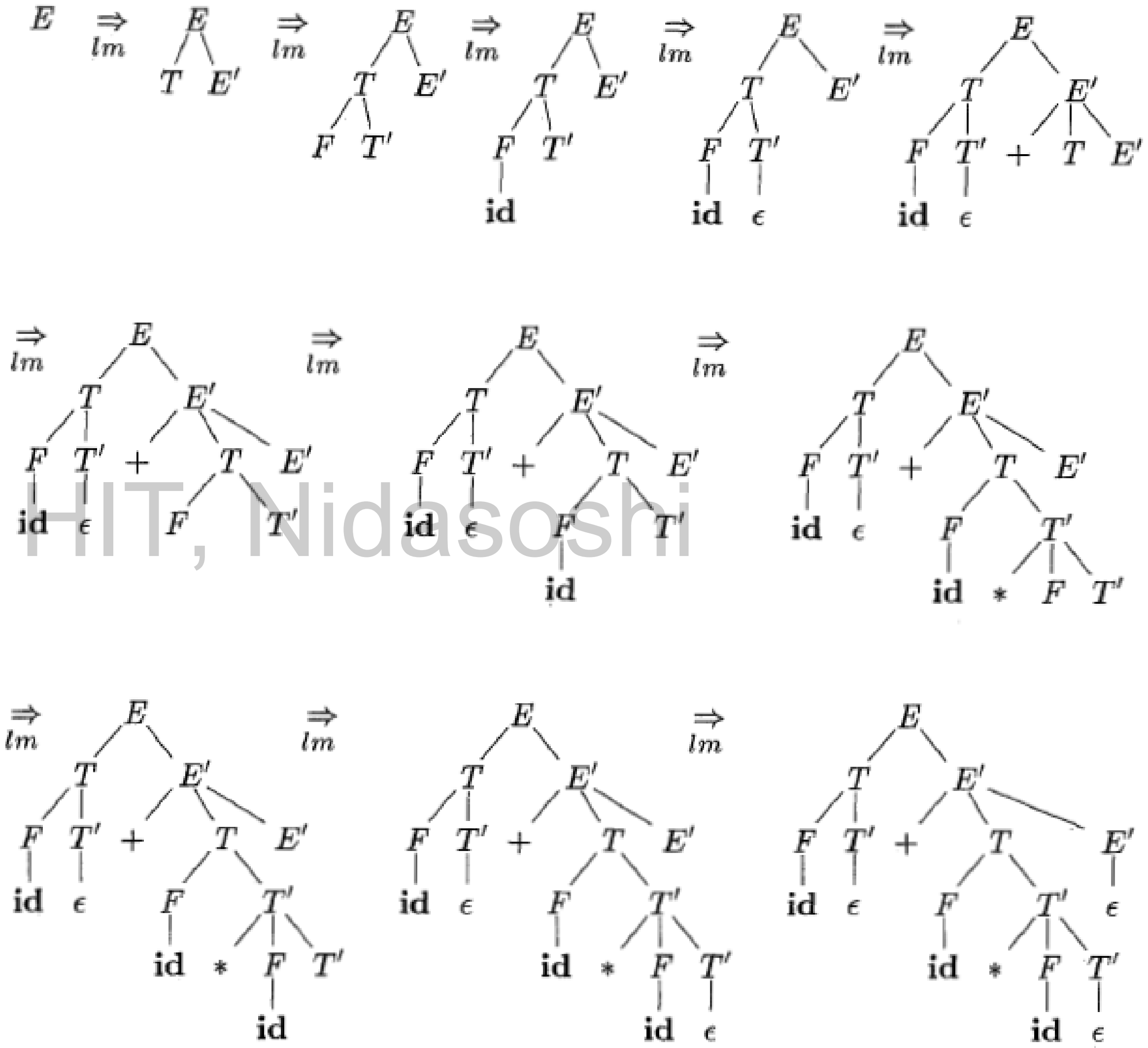
Top-Down Parsing

- Input String: **id+id*id**

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \epsilon \\
F &\rightarrow (\ E\ ) \mid \mathbf{id}
\end{aligned}
$$

- Input String: **id+id*id**

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \textbf{id}
\end{array}
$$



Top-Down Parsing - Recursive-Descent Parsing

- A recursive-descent parsing program consists of a set of procedures, one for each nonterminal.
- Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.
- Pseudocode for a typical nonterminal appears in Fig

```
void A() {
    Choose an A-production, A -> X1 X2 ... Xk;
    for ( i = 1 to k ) {
        if ( Xi is a nonterminal )
            call procedure Xi();
        else if ( Xi equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}
```

Top-Down Parsing - Recursive-Descent Parsing

- General recursive-descent may require backtracking; that is, it may require repeated scans over the input.
- However, backtracking is rarely needed to parse programming language constructs, so backtracking parsers are not seen frequently.
- Even for situations like natural language parsing, backtracking is not very efficient, and tabular methods are preferred.

Top-Down Parsing - Recursive-Descent Parsing

- Consider the grammar

$$\begin{aligned} S &\rightarrow c\ A\ d \\ A &\rightarrow a\ b \mid a \end{aligned}$$

- 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.

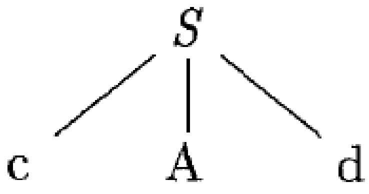
- *S* has only one production, so we use it to expand S and obtain the tree of Fig (a).
- The leftmost leaf, labeled c, matches the first symbol of input w, so we advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A.

Top-Down Parsing - Recursive-Descent Parsing

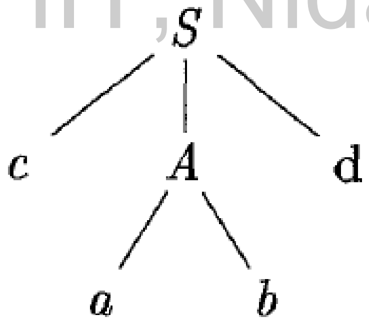
- Now, we expand A using the first alternative A -+ a *b* to obtain the tree of Fig. (b).
- We have a match for the second input symbol, a, so we advance the input pointer to d, the third input symbol, and compare d against the next leaf, labeled b.
- Since *b* does not match d, we report failure and 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, which means that the procedure for A must store the input pointer in a local variable.

Top-Down Parsing - Recursive-Descent Parsing

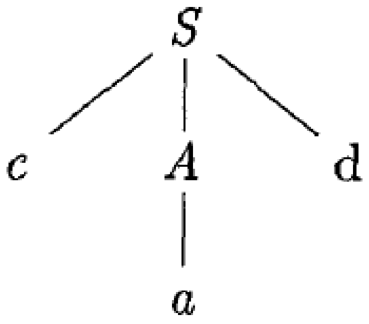
- The second alternative for A produces the tree of Fig. (c).
- The leaf a matches the second symbol of w and the leaf d matches the third symbol.
- Since we have produced a parse tree for w, we halt and announce successful completion of parsing.



(a)



(b)



(c)

Top-Down 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 topdown parsing, FIRST and FOLLOW allow us to choose which production to
- apply, based on the next input symbol.
- During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

Top-Down Parsing - FIRST and FOLLOW

- To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or **E:** can be added to any FIRST set.

1. If X is a terminal, then FIRST$(X) = \{X\}$.

2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ 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$(Y_1), \ldots,$ FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST$(Y_2)$, and so on.

3. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

Top-Down Parsing - FIRST and FOLLOW

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E'\ |\ \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T'\ |\ \epsilon \\
F & \rightarrow & (\ E\ )\ |\ \mathbf{id}
\end{array}
$$

1. FIRST(F) = FIRST(T) = FIRST(E) = **{(, id}.** To see why, note that the two productions for F have bodies that start with these two terminal symbols, **id** and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive **ε**, FIRST(T) must be the same as FIRST(F). The same argument covers FIRST(E).

2. FIRST(E') = {+, ε}. The reason is that one of the two productions for E' has a body that begins with terminal +, and the other's body is E. Whenever a nonterminal derives E, we place E in FIRST for that nonterminal.

3. FIRST(T') = {*, ε}. The reasoning is analogous to that for FIRST(E').

Top-Down Parsing - FIRST and FOLLOW

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E'\ |\ \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T'\ |\ \epsilon \\
F & \rightarrow & (\ E\ )\ |\ \mathbf{id}
\end{array}
$$

- FIRST(F) = FIRST(T) = FIRST(E) = **{(, id}.** To see why, note that the two productions for F have bodies that start with these two terminal symbols, **id** and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive **ε**, FIRST(T) must be the same as FIRST(F). The same argument covers FIRST(E).

- FIRST(E') = {+, ε}. The reason is that one of the two productions for E' has a body that begins with terminal +, and the other's body is E. Whenever a nonterminal derives E, we place E in FIRST for that nonterminal.

- FIRST(T') = {*, ε}. The reasoning is analogous to that for FIRST(E').

Top-Down Parsing - FIRST and FOLLOW

- To compute FOLLOW(A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set.

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

2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).

3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Top-Down Parsing - FIRST and FOLLOW

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \mathbf{id} \end{array}$$

- FOLLOW(E) = FOLLOW(E') = {), $ }. Since E is the start symbol, FOLLOW(E) must contain $. The production body (E) explains why the right parenthesis is in FOLLOW(E). For E', note that this nonterminal appears only at the ends of bodies of E-productions. Thus, FOLLOW(E') must be the same as FOLLOW(E).

Top-Down Parsing - FIRST and FOLLOW

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \textbf{id}
\end{array}
$$

- FOLLOW(T) = FOLLOW(T') = {+,), **$}.** Notice that T appears in bodies only followed by E'. Thus, everything except **ε** that is in FIRST(E') must be in FOLLOW (T) ; that explains the symbol +. However, since FIRST(E') contains **ε**, and E' is the entire string following T in the bodies of the E-productions, everything in FOLLOW(E') must also be in FOLLOW(T). That explains the symbols $ and the right parenthesis.
- As for T', since it appears only at the ends of the T-productions, it must be that FOLLOW(T') = FOLLOW(T)

Top-Down Parsing - FIRST and FOLLOW

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E'\ |\ \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T'\ |\ \epsilon \\
F & \rightarrow & (\ E\ )\ |\ \mathbf{id}
\end{array}
$$

- FOLLOW(F) = {+, *,), $}. The reasoning is analogous to that for T in point.

Top-Down Parsing - 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 first "L" in LL(1) stands for scanning the input from **left to right**, the second "L" for producing a **leftmost derivation**, and the "1" for using one input symbol of lookahead at each step to make parsing action decisions.
- The class of LL(1) grammars is rich enough to cover most programming constructs, although care is needed in writing a suitable grammar for the source language. For example, no left-recursive or ambiguous grammar can be LL(1)

Top-Down Parsing – Predictive Parsing Table

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

- If, after performing the above steps, there is no production at all in M[A, a], then set M[A, a] to error.

Top-Down Parsing – Predictive Parsing Table

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

- If, after performing the above steps, there is no production at all in M[A, a], then set M[A, a] to error.

Top-Down Parsing – Predictive Parsing Table

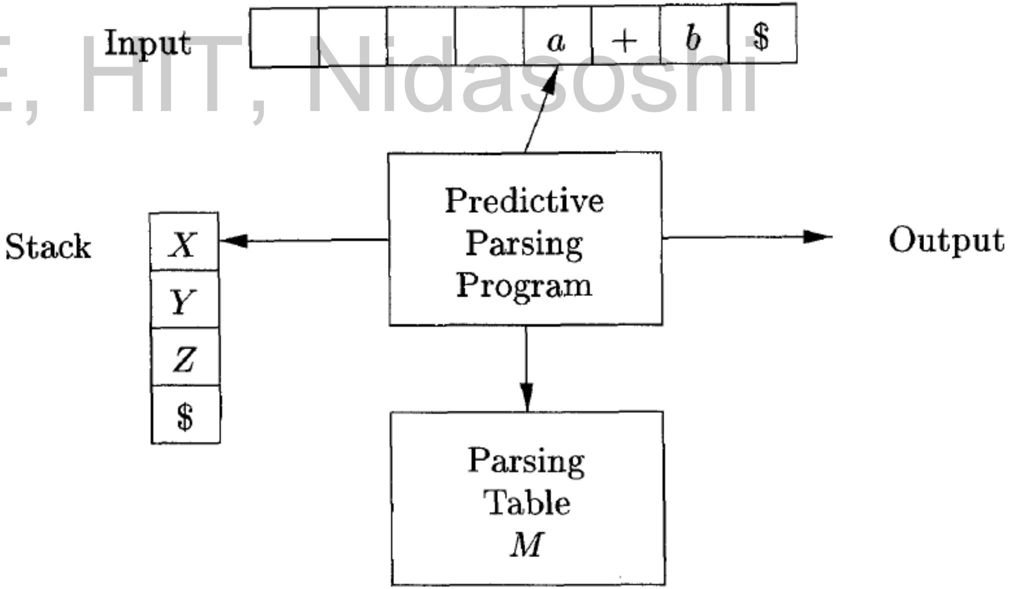
$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

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

Top-Down Parsing – Nonrecursive (Table-driven) Predictive Parsing

- A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls. The parser mimics a leftmost derivation.
- If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols *a* such that

$$S \overset{*}{\underset{lm}{\Rightarrow}} w\alpha$$



Top-Down Parsing – Nonrecursive (Table-driven) Predictive Parsing

- The table-driven parser in previous Fig. has an input buffer, a stack containing a sequence of grammar symbols, a parsing table, and an output stream.
- The input buffer contains the string to be parsed, followed by the endmarker $. We reuse the symbol $ to mark the bottom of the stack, which initially contains the start symbol of the grammar on top of $.

- The parser is controlled by a program that considers X, the symbol on top of the stack, and a, the current input symbol.
- If X is a nonterminal, the parser chooses an X-production by consulting entry M[X, a] of the parsing table M.
- Otherwise, it checks for a match between the terminal X and current input symbol a.

Top-Down Parsing – Nonrecursive (Table-driven) Predictive Parsing

- Example input string:
- **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 $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	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 $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	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 $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Top-Down Parsing – Nonrecursive (Table-driven) Predictive Parsing

- 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.

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X is a ) pop the stack and advance ip;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1Y2 ··· Yk ) {
                output the production X → Y1Y2 ··· Yk;
                pop the stack;
                push Yk, Yk-1, ... , Y1 onto the stack, with Y1 on top;
        }
        set X to the top stack symbol;
}
```

Top-Down Parsing – Error Recovery in Predictive Parsing

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

- There are two ways to recover form error.

1. Panic Mode Recovery
2. Phrase Level Error Recovery

Top-Down Parsing – Error Recovery in Predictive Parsing

- **Panic Mode Recovery**
- Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears.
- Its effectiveness depends on the choice of synchronizing set.
  - Usually, we use **FOLLOW** symbols as synchronizing tokens
  - Use synch in predictive parse table to indicate the synchronizing token obtained from FOLLOW SET of the non-terminal.

Top-Down Parsing – Error Recovery in Predictive Parsing

- **Panic Mode Recovery - Rules**

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

Top-Down Parsing – Error Recovery in Predictive Parsing

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E'\ \mid\ \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T'\ \mid\ \epsilon \\
F &\rightarrow (\ E\ )\ \mid\ \mathbf{id}
\end{aligned}
$$

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

Synchronizing tokens added to the parsing table

Top-Down Parsing – Error Recovery in Predictive Parsing

STACK	INPUT	REMARK
$E\,\$$	$)\,\mathbf{id} * + \mathbf{id}\,\$$	error, skip)
$E\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	**id** is in FIRST(E)
$TE'\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$FT'E'\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$\mathbf{id}\ T'E'\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$T'E'\,\$$	$* + \mathbf{id}\,\$$	
$* FT'E'\,\$$	$* + \mathbf{id}\,\$$	
$FT'E'\,\$$	$+ \mathbf{id}\,\$$	error, $M[F,+]$ = synch
$T'E'\,\$$	$+ \mathbf{id}\,\$$	F has been popped
$E'\,\$$	$+ \mathbf{id}\,\$$	
$+ TE'\,\$$	$+ \mathbf{id}\,\$$	
$TE'\,\$$	$\mathbf{id}\,\$$	
$FT'E'\,\$$	$\mathbf{id}\,\$$	
$\mathbf{id}\,T'E'\,\$$	$\mathbf{id}\,\$$	
$T'E'\,\$$	$\$$	
$E'\,\$$	$\$$	
$\$$	$\$$	

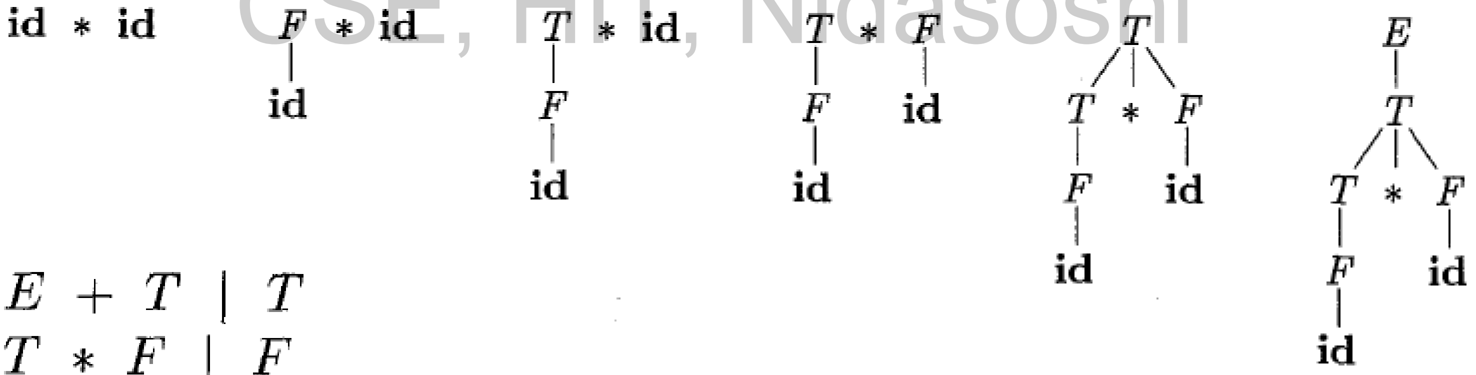
Top-Down Parsing – Error Recovery in Predictive Parsing

Phrase-level Recovery

- Phrase-level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines.
- These routines may change, insert, or delete symbols on the input and issue appropriate error messages.

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).
- It is convenient to describe parsing as the process of building parse trees, although a front end may in fact carry out a translation directly without building an explicit tree.



$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

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

Bottom-Up Parsing – 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 key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

$$\mathbf{id} * \mathbf{id}, \quad F * \mathbf{id}, \quad T * \mathbf{id}, \quad T * F, \quad T, \quad E$$

Bottom-Up Parsing – Reductions

$$\mathbf{id} * \mathbf{id}, \quad F * \mathbf{id}, \quad T * \mathbf{id}, \quad T * F, \quad T, \quad E$$

- The strings in this sequence are formed from the roots of all the subtrees in the snapshots. The sequence starts with the input string id*id.
- The first reduction produces F * id by reducing the leftmost id to F, using the production F -+ id.
- The second reduction produces T * id by reducing F to T.
- Now, we have a choice between reducing the string T, which is the body of E -+ T, and the string consisting of the second id, which is the body of F -+ id.
- Rather than reduce T to E, the second id is reduced to T, resulting in the string T * F.
- This string then reduces to T.
- The parse completes with the reduction of T to the start symbol E.

Bottom-Up Parsing – Handle Pruning

- Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse.
- Informally, 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.
- For example, adding subscripts to the tokens **id** for clarity, the handles during the parse of **idl * id2.**

- Although T is the body of the production E → T, the symbol T is not a handle in the sentential form T * id2.
- If T were indeed replaced by E, we would get the string E * id2, which cannot be derived from the start symbol E.
- Thus, the leftmost substring that matches the body of some production need not be a handle.

Bottom-Up Parsing – Handle Pruning

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

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

Bottom-Up Parsing – Handle Pruning

- S → aABc
- A → Abc | b
- B → d

- Input String: **abbcde**

Bottom-Up Parsing – Shift-Reduce Parsing

- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- As we shall see, the handle always appears at the top of the stack just before it is identified as the handle.
- We use $ to mark the bottom of the stack and also the right end of the input.
- Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right, rather than on the left as we did for top-down parsing.

Bottom-Up Parsing – Shift-Reduce Parsing

- Initially, the stack is empty, and the string **w** is on the input, as follows:

STACK	INPUT
$	w $

- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string *P* of grammar symbols on top of the stack.

- It then reduces ,O to the head of the appropriate production.
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

STACK	INPUT
$ S	$

Bottom-Up Parsing – Shift-Reduce Parsing

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \,\$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \,\$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \,\$$	reduce by $T \to F$
$\$\,T$	$* \mathbf{id}_2 \,\$$	shift
$\$\,T *$	$\mathbf{id}_2 \,\$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

$$
\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (E) \mid \mathbf{id}
\end{aligned}
$$

Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

Bottom-Up Parsing – Shift-Reduce Parsing

While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.

1. ***Shift.*** Shift the next input symbol onto the top of the stack.
2. ***Reduce***. The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. ***Accept***. Announce successful completion of parsing.
4. ***Error***. Discover a syntax error and call an error recovery routine.

Bottom-Up Parsing – Shift-Reduce Parsing

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Input String: **(id) + id**

Bottom-Up Parsing – Shift-Reduce Parsing

S → 0S) | 1S1 | 2

Input string: 10201

Bottom-Up Parsing – Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsing cannot be used.
- Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide **whether to shift or to reduce (a shift/reduce conflict), or cannot decide which of several reductions to make (a *reduce/reduce conflict*).**
- Two types of Conflicts:
  1. Shift-Reduce Conflict
  2. Reduce-Reduce Conflict

Bottom-Up Parsing – Conflicts During Shift-Reduce Parsing

- Shift-Reduce Conflict
  - Whether to shift the next input symbol or reduce the current handle
- Example:
- S → AB
- A → 0S | 1S
- B → 0S1 | 1S1
- Input String: **0S1S1**

Bottom-Up Parsing – Conflicts During Shift-Reduce Parsing

- Reduce-Reduce Conflict
  - During Parsing with known stack contents and the next input symbol, the parser identifies the handle on the top of stack (TOS), the parser can reduce the handle by applying production. But there is a possibility to apply one more production to the same handle. So, the parser cannot decide which production to apply to reduce the handle. That is **“which of the several production to apply”**

- Example:
- S → AB
- A → 0S | 1S
- B → 0S1 | 1S
- Input String: **0S1S**

CSE, HIT, Nidasoshi

Lexical Versus Syntactic Analysis

1. Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.