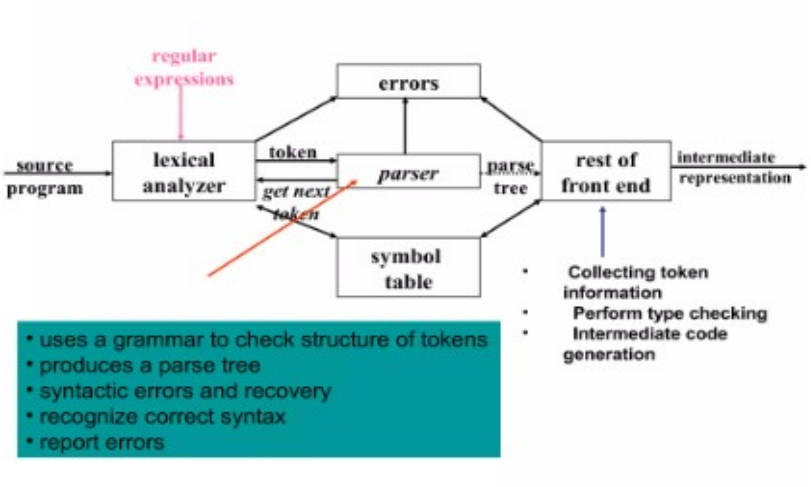
# UNIT-2

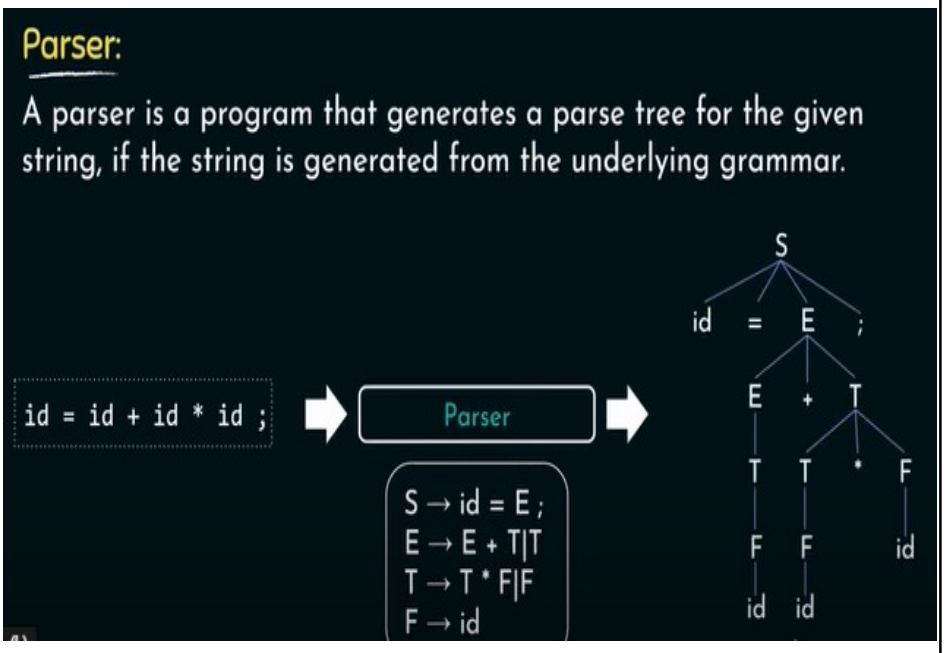
- **Syntax Analysis:**
- **Introduction:** The Role of the parser – Representative Grammars – Syntax Error Handling – Error Recovery Strategies.
- **Context Free Grammars:** The formal definition of a CFG – Notational Conventions – Derivations – Parse trees and derivations – Ambiguity
- **Writing Grammar:** Lexical Versus Syntax Analysis – Eliminating Ambiguity – Elimination of Left Recursion – Left Factoring
- **Top Down Parsing:** Recursive Descent Parsing-FIRST and FOLLOW - LL(1) Grammars – Non recursive Predictive Parsing- Error Recovery in Predictive Parsing.

# The Role of the parser



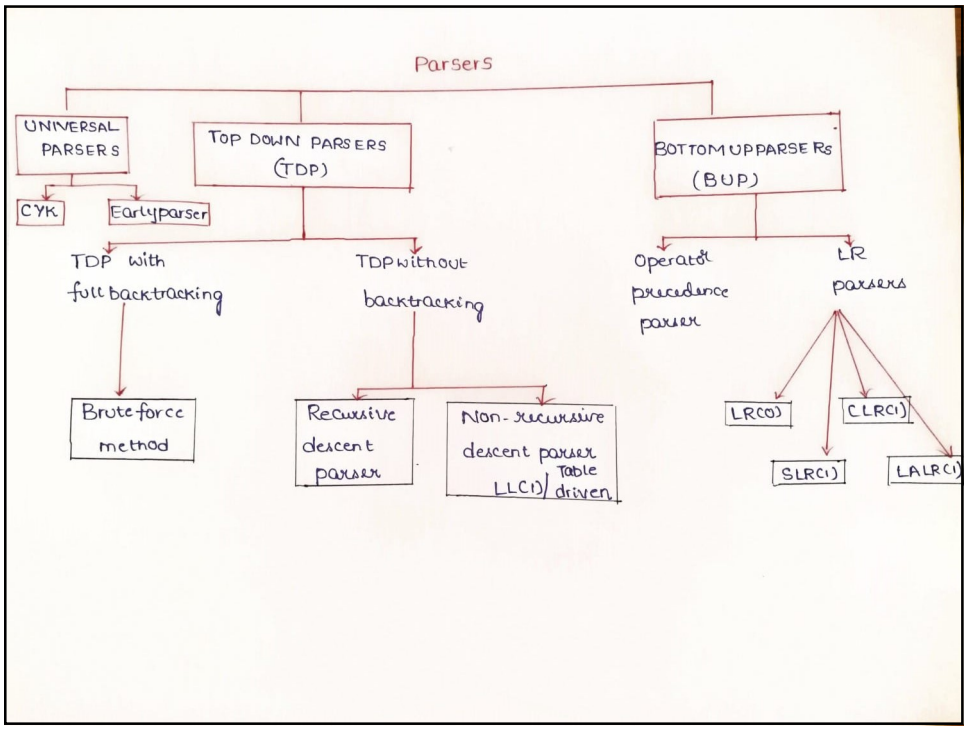
# The Role of the parser

- The parser obtains a string of tokens from lexical analyzer.
- It then verifies whether the input string can be generated from the grammar of the source language.
- It has to report any syntactical errors and recover from commonly occurring error to continue the processing the remainder of the program
- The parser constructs a parse tree and passes it to the rest of the compiler for further processing.
- It should also report syntactical errors in an easily understandable way to the user.



# Categorization of parsers

- There are 03 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. These general methods are, however, too inefficient to use in production compilers.
  - The methods commonly used in compilers is either top-down or bottom-up.
  - Top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root.



# Representative Grammars

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

Useful for Bottom up parsing. Belongs to LR grammar Class.

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

After removing left recursion used for top down parsing

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

useful for illustrating handling ambiguities during parsing

# Syntax Error Handling

Common programming errors can occur at many different levels

**Types of Errors**

*1) Lexical*

*2) Syntactic*

3) Semantic

4) Logical

- **Lexical errors** include misspellings of identiers, keywords, or operators.
  - e.g., the use of an identier *elipseSize* instead of *ellipseSize*
  - missing quotes around text intended as a string
- **Syntactic errors** include misplaced semicolons or extra or missing braces; that is, "{"or "}"
  - Example: In C or Java, the appearance of a *case* statement without an enclosing *switch* is a syntactic error

## Syntax Error Handling

Types of Errors

1) Lexical
2) Syntactic
*3) Semantic*
*4) Logical*

- **Semantic errors** include type mismatches between operators and operands,
  - e.g., the return of a value in a Java method with result type void.
- **Logical errors** occur when executed code does not produce the expected result.
  - incorrect reasoning on the part of the programmer
  - 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

1. Panic-Mode Recovery
2. Phrase-Level Recovery
3. Error Productions
4. Global Correction

## Panic-Mode Recovery

- Once an error is found, the parser intends to find designated set of synchronizing tokens by discarding input symbols one at a time.
- Synchronizing tokens are *delimiters, semicolon or }* whose role in source program is clear.
- When parser finds an error in the statement, it ignores the rest of the statement by not processing the input.
- *Advantage:*
  - Simplicity
  - Never get into infinite loop
- *Disadvantage:*
  - Additional errors cannot be checked as some of the input symbols will be skipped.

## Phrase-Level Recovery

- When a parser finds an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. The corrections may be
  - Replacing a prefix by some string.
  - Replacing comma by semicolon.
  - Deleting extraneous semicolon.
  - Inserting missing semicolon.

we must be careful to choose replacements that do not lead to infinite loops, as would be the case, for example, if we always inserted something on the input ahead of the current input symbol.

- *Advantage:*
  - It can correct any input string.
- *Disadvantage:*
  - It is difficult to cope up with actual error if it has occurred before the point of detection.

## Error Productions

- The use of the error production method can be incorporated if the user is aware of common mistakes that are encountered in grammar in conjunction with errors that produce erroneous constructs.
  - Example: write 5x instead of 5*x
- *Advantage:*
  - If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.
- *Disadvantage:*
  - The disadvantage is that it's difficult to maintain.

## Global Correction

- The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free.
- When an erroneous input statement X is fed, it creates a parse tree for some closest error-free statement Y.
- *Advantage:*
  - This may allow the parser to make minimal changes in the source code.
- *Disadvantage:*
  - Due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

## Lexical analysis Vs syntax analysis

1. Separating the syntactic structure of a language into lexical and non-lexical 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-under-stand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

# Lexical Vs Syntactic analysis

| Lexical analysis | Syntax analysis |
|---|---|
| It is responsible for converting a sequence of characters into a pattern of tokens. | Process of analyzing a string of symbols either in natural language or computer languages that satisfies the rules of a formal grammar. |
| Reads the program one character at a time, the output is meaningful lexemes. | Tokens are taken as input and a parse tree is generated as output. |
| It is the first phase of the compilation process. | It is the second phase of the compilation process. |

| Lexical analysis | Syntax analysis |
|---|---|
| It can also be referred to as lexing and tokenization. | It can also be referred to as syntactic analysis and parsing. |
| A lexical analyser is a pattern matcher. | A syntax analysis involves forming a tree to identify deformities in the syntax of the program. |
| Less complex approaches are often used for lexical analysis. | Syntax analysis requires a much more complex approach. |

# Context Free Grammars

- The formal definition of a CFG
- Notational Conventions
- Derivations
- Parse trees and derivations
- Ambiguity

# The formal definition of a CFG

- Context-free grammars are named as such because **any of the production rules in the grammar can be applied regardless of context**—it does not depend on any other symbols that may or may not be around a given symbol that is having a rule applied to it.
- A context free grammar G is defined by four tuple format as

$$G = (V, T, P, S)$$

where,

G – Grammar
V – Set of variables
T – Set of terminals
P – Set of productions
S – Start symbol

# A Context Free Grammar

- A context-free grammar has four components:
  - A set of terminal symbols, sometimes referred to as "tokens."
  - A set of nonterminals, sometimes called "syntactic variables."
  - A set of productions, where each production consists of a nonterminal,called the head or left side of the production, an arrow, and a sequence of terminals and/or nonterminals , called the body or right side of the production
  - A designation of one of the nonterminals as the start symbol.

Example 1:

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

the terminal symbols are **id + - * / ( )**

The nonterminal symbols are *expression*, *term* and *factor*,

*expression* is the start symbol

Example 2:

*stmt* → **if** ( *expr* ) *stmt* **else** *stmt*

# Notational Conventions

1. These symbols are terminals:
   (a) Lowercase letters early 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 $0, 1, \ldots, 9$.
   (e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.

2. These symbols are nonterminals:
   (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 nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by $E$, $T$, and $F$, respectively.

3. Uppercase letters late in the alphabet, such as $X$, $Y$, $Z$, represent *grammar symbols*; that is, either nonterminals or terminals.

4. Lowercase letters late in the alphabet, chiefly $u, v, \ldots, z$, represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, $\alpha$, $\beta$, $\gamma$ for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as $A \rightarrow \alpha$, where $A$ is the head and $\alpha$ the body.

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

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

- Example

E → E + T | E − T | T
T → T * F | T / F | F
F → (E) | *id*

**V** = {E, T, F}
**T** = {+ , −, *, /, ( , ), *id*}
**S** = {E}
**P** :

| | |
|---|---|
| E → E+T | T→ T / F |
| E → E−T | T→ F |
| E →T | F→ (E) |
| T → T * F | F→ *id* |

# Derivations

- $E \Rightarrow E+E$ : E+E derives from E
- $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$
- A sequence of replacements of non-terminal symbols is called a **derivation** of id+id from E.
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if there is a production rule $A \rightarrow \gamma$ in our grammar and $\alpha$ and $\beta$ are arbitrary strings of terminal and non-terminal symbols

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ ($\alpha_n$ derives from $\alpha_1$ or $\alpha_1$ derives $\alpha_n$)

$\Rightarrow$ : derives in one step

$\overset{*}{\Rightarrow}$ : derives in zero or more steps

$\overset{+}{\Rightarrow}$ : derives in one or more steps

# CFG - Terminology

- 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.
- If S is the start symbol of G then

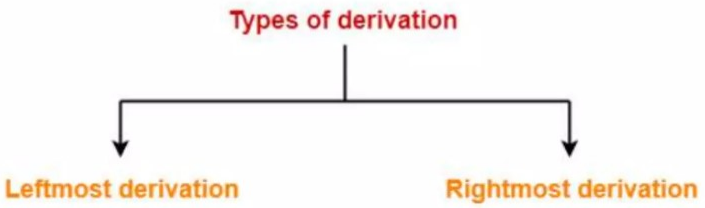
  $\omega$ is a sentence of L(G) iff $S \overset{*}{\Rightarrow} \omega$ where $\omega$ is a string of terminals of G
- If G is a context-free grammar, L(G) is a *context-free language*.
- Two grammars are *equivalent* if they produce the same language.
- $S \overset{*}{\Rightarrow} \alpha$ - If $\alpha$ contains non-terminals, it is called as a *sentential* form of G.

  - If $\alpha$ does not contain non-terminals, it is called as a *sentence* of G.

# Derivations

**Parse Tree-**

- The process of deriving a string is called as **derivation**.
- The geometrical representation of a derivation is called as a **parse tree** or **derivation tree.**




# LEFT MOST DERIVATION

At each and every step the leftmost non-terminal is expanded by substituting its corresponding production to derive a string.

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

$w = \mathbf{id} + \mathbf{id} * \mathbf{id}$

$E \underset{lm}{\rightarrow} E + E$

$E \underset{lm}{\rightarrow} id + E \quad [E \rightarrow id]$

$E \underset{lm}{\rightarrow} id + E * E \quad [E \rightarrow E * E]$

$E \underset{lm}{\rightarrow} id + id * E \quad [E \rightarrow id]$

$E \underset{lm}{\rightarrow} id + id * id \quad [E \rightarrow id]$

# RIGHT MOST DERIVATION

At each and every step the rightmost non-terminal is expanded by substituting its corresponding production to derive a string.

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

$w = \mathbf{id} + \mathbf{id} * \mathbf{id}$

$E \underset{rm}{\rightarrow} E + E$

$E \underset{rm}{\rightarrow} E + E * E \quad [E \rightarrow E * E]$

$E \underset{rm}{\rightarrow} E + E * \mathbf{id} \quad [E \rightarrow \mathrm{id}]$

$E \underset{rm}{\rightarrow} E + \mathbf{id} * \mathbf{id} \quad [E \rightarrow \mathrm{id}]$

$E \rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \quad [E \rightarrow \mathrm{id}]$

# EXAMPLE

- Leftmost

$S \rightarrow SS+ \mid SS* \mid a$

$w = aa + a*$

$S \underset{lm}{\rightarrow} SS*$

$S \underset{lm}{\rightarrow} SS + S* \quad [S \rightarrow SS+]$

$S \underset{lm}{\rightarrow} aS + S* \quad [S \rightarrow a]$

$S \underset{lm}{\rightarrow} aa + S* \quad [S \rightarrow a]$

$S \rightarrow aa + a* \quad [S \rightarrow a]$

- Rightmost

$S \rightarrow SS+ \mid SS* \mid a$

$w = aa + a*$

$S \underset{rm}{\rightarrow} SS*$

$S \underset{rm}{\rightarrow} Sa* \quad [S \rightarrow a]$

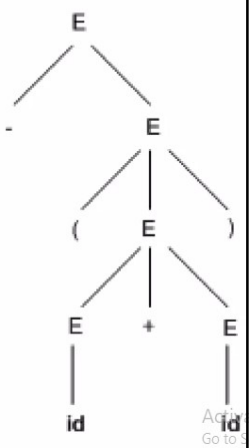
$S \underset{rm}{\rightarrow} SS + a* \quad [S \rightarrow SS+]$

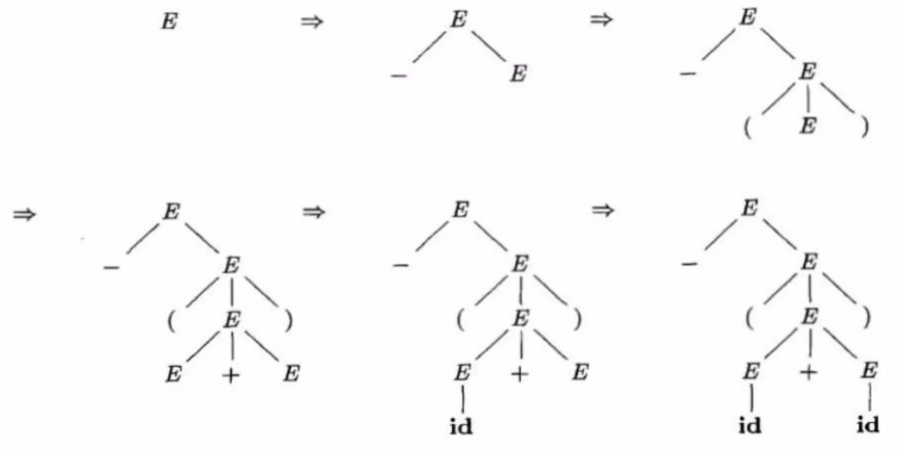
$S \underset{rm}{\rightarrow} Sa + a* \quad [S \rightarrow a]$

$S \underset{rm}{\rightarrow} aa + a* \quad [S \rightarrow a]$

# Parse Tree

- Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings.
- Root node of parse tree has the start symbol of the given grammar from where the derivation proceeds.
- Leaves of parse tree represent terminals.
- Each interior node represents productions of grammar.
- If G : E → E+E | E*E | ( E ) | - E | id is the grammar, then Parse tree for the input string - **(id + id)** is shown.

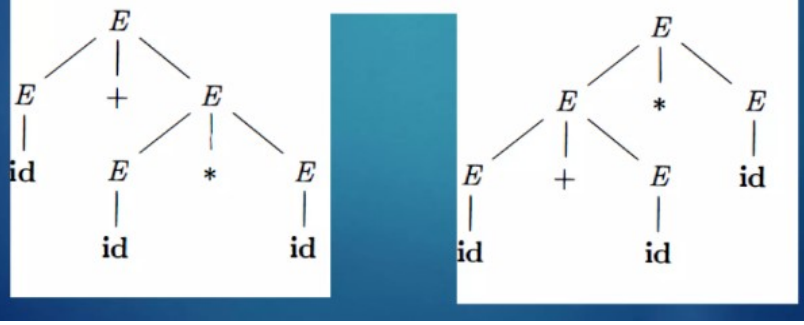


# Ambiguity

- a grammar that produces more than one parse tree for some sentence is said to be ambiguous

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




# Assignment 1

**Exercise 4.2.1:** Consider the context-free grammar:

$$S \rightarrow S\,S\,+ \mid S\,S\,* \mid a$$

and the string $aa + a*$.

a) Give a leftmost derivation for the string.

b) Give a rightmost derivation for the string.

c) Give a parse tree for the string.

! d) Is the grammar ambiguous or unambiguous? Justify your answer.

! e) Describe the language generated by this grammar.

1. Give a leftmost derivation for the string.
2. Give a rightmost derivation for the string.
3. Give a parse tree for the string.
4. Is the grammar ambiguous or unambiguous?

a. $S \rightarrow 0\ S\ 1 \mid 0\ 1$ with string 000111.

b. $S \rightarrow +\ S\ S \mid *\ S\ S \mid a$ with string $+ * aaa$.

c. $S \rightarrow S\ (\ S\ )\ S \mid \epsilon$ with string (()()).

d. $S \rightarrow S\ +\ S \mid S\ S \mid (\ S\ ) \mid S\ * \mid a$ with string $(a + a) * a$.

e. $S \rightarrow (\ L\ ) \mid a$ and $L \rightarrow L\ ,\ S \mid S$ with string $((a, a), a, (a))$.

f. $S \rightarrow a\ S\ b\ S \mid b\ S\ a\ S \mid \epsilon$ with string $aabbab$.

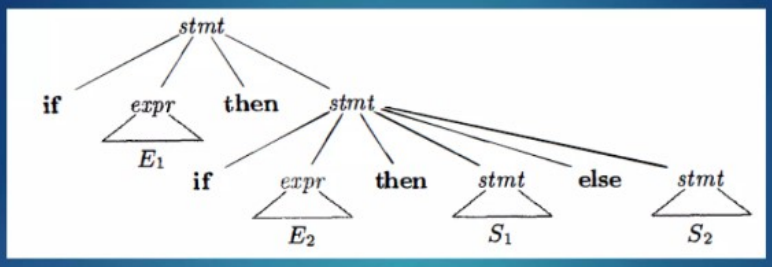
# Writing a Grammar

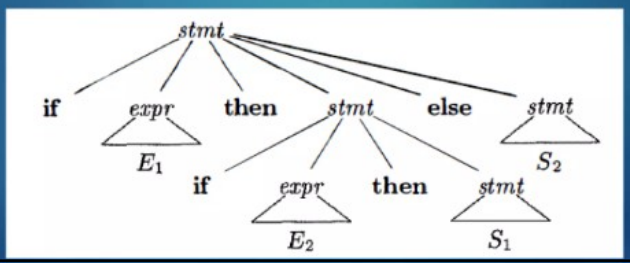
- Grammars are capable of describing most, of the syntax of programming languages .
- Grammar should be unambiguous.
- Left-recursion elimination and left factoring - are useful for rewriting grammars .
- From the resulting grammar we can create top down parsers without backtracking.
- Such parsers are called predictive parsers or recursive-descent parser

# Eliminating Ambiguity

- ambiguous grammar can be rewritten to eliminate the ambiguity.
- stmt -> if expr then stmt
  |if expr then stmt else stmt
  |other
- is ambiguous since the string
  - if E1 then if E2 then S1 else S2 has the two parse trees

# Two parse trees for an ambiguous sentence



# Eliminating Ambiguity

- The general rule is, "Match each else with the closest unmatched then."

| | | |
|---|---|---|
| *stmt* | → | *matched_stmt* |
| | \| | *open_stmt* |
| *matched_stmt* | → | **if** *expr* **then** *matched_stmt* **else** *matched_stmt* |
| | \| | **other** |
| *open_stmt* | → | **if** *expr* **then** *stmt* |
| | \| | **if** *expr* **then** *matched_stmt* **else** *open_stmt* |

# Left Recursion

- 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 $\alpha$

- Top-down parsing techniques cannot handle left-recursive grammars.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

## Immediate Left-Recursion

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

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\,A'$

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

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\,A' \mid \ldots \mid \beta_n\,A'$

$A' \rightarrow \alpha_1\,A' \mid \ldots \mid \alpha_m\,A' \mid \varepsilon$ an equivalent grammar

- Here is an example of a (directly) left-recursive grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow ( E ) \mid id$

- This grammar can be re-written as the following non left-recursive grammar:

$E \rightarrow T\,E'$ $E' \rightarrow + TE' \mid \epsilon$

$T \rightarrow F\,T'$ $T' \rightarrow * F\,T' \mid \epsilon$

$F \rightarrow (E) \mid id$

# Left Factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing.
- Stmt -> if expr then stmt else stmt
  |if expr then stmt
- A ->α $\beta_1$ | α $\beta_2$
- So it should be left factored as

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

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$$

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

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

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

# Left-Factoring -- Algorithm

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix

$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$

convert it into

$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$

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

# Left-Factoring – Example1

$A \rightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid cdg \mid cdeB \mid cdfB$

$A' \rightarrow bB \mid B$

$\Downarrow$

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

$A' \rightarrow bB \mid B$

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

# Left-Factoring – Example2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

# Left-Recursion -- Problem

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

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$

$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

## Removing left recursion: Example 1

- S-> Sca/b/da
- A->Aac/d/bc

S->bS'/daS'
S'->caS'/€

A->dA'/bcA'
A'->acA'/€

## Example 2

- E->EZM/M/2M
- M->MUF/F

  - E->ME'/2ME'
  - E'->ZME'/€

  - M->FM'
  - M'->UFM'/€

## Example 3

- S->S;S/id:=E/print(L)
  - S->id:=ES'/print(L)S'
  - S'->;SS'/€
- E->E+E/id/num/(S,E)

  E->idE'/numE'/(S,E)E'

  E'->+EE'/€
- L->E/L,E

  L->EL'

  L'->,EL'/€

## Remove left factoring/left recursion

- D->X/X,D
- X->id/id[c]
- S->T/T;S
- A->id=E/id(E)=E
- F->C/id/id[E]
- C->G/GC
- S->iEtS/iEtSeS

# Types of parsing techniques

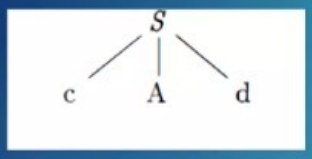
- Top down parsing and bottom up parsing
- Top down parsing:
  - Recursive Descent parsing(or) brute force method
  - Predictive parsing(or) non recursive descent parsing

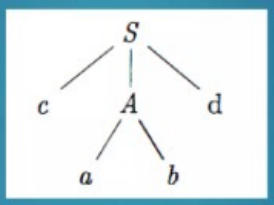
# Top-Down Parsing

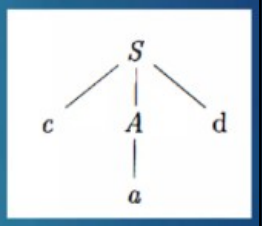
- The parse tree is created top to bottom.
- Top-down parser
  - Recursive-descent parsing
    - Backtracking is needed
    - It is a general parsing technique, but not widely used.
    - Not efficient
  - Predictive parsing
    - No backtracking
    - Efficient
    - Needs a special form of grammars - (LL(1) grammars).
    - Recursive predictive parsing is a special form of recursive descent parsing without backtracking.
    - Non-recursive (table driven) predictive parser is also known as LL(1) parser.

## Recursive Descent parsing : string "cad"

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




## Examples for recursive descent parsing

- S->abA
- A->cd/c/€

derive string "ab"

- S->abA
- A->bc/b

derive string"abb"

# Predictive parsing

- This is also called as recursive predictive parsing
- No need of backtracking
- To construct predictive parsing, we need to compute the First and Follow for all Non terminals

- Predictive parsing
  - No backtracking
  - Efficient
  - Needs a special form of grammars - (LL(1) grammars).
  - Recursive predictive parsing is a special form of recursive descent parsing without backtracking.
  - Non-recursive (table driven) predictive parser is also known as LL(1) parser.

## RULES FOR FIRST SET

1. If X is terminal, then FIRST(X) is {X}.
2. If X => ε is a production, then add ε to FIRST(X).
3. If X is nonterminal and X => $Y_1 Y_2 \dots Y_k$ is a production, 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($Y_{i-1}$); that is, $Y_1, \dots, Y_{i-1}$ => ε. If ε is in FIRST($Y_j$) for all $j = 1, 2, \dots, k$, 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 $Y_1$ => ε, then we add FIRST($Y_2$) and so on.

# RULES FOR FOLLOW SET

1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right end marker.
2. If there is a production A => α Bβ, then everything in FIRST(β), except for Ɛ, is placed in FOLLOW(B).
3. If there is a production A => α B, or a production A => α Bβ where FIRST(β) contains Ɛ (i.e., β => Ɛ), then everything in FOLLOW(A) is in FOLLOW(B).

# EXAMPLES

E => T E'

E'=> + T E' | Ɛ

T => F T'

T'=> * F T' | Ɛ

F => ( E ) | id

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

FIRST(E') = {+, Ɛ}

FIRST(T') = {*, Ɛ}

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

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

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

# EXAMPLE

S => A a

A => B D

B => b | ε

D => d | ε

First(S) = {b, d, a}

First(A) = {b, d, ε}

First(B) = {b, ε}

First(D) = {d, ε }

Follow(S) = {$}

Follow(A) = {a}

Follow(B) = {d, a}

Follow(D) = {a}

1. C => P F class id X Y
2. P => public
3. P => ε
4. F => final
5. F => ε
6. X => extends id
7. X => ε
8. Y => implements I
9. Y => ε
10. I => id J
11. J => , I
12. J => ε

First(C) = {public, final, class}

First(P) = {public, ε }

First(F) = {final, ε }

First(X) = {extends, ε }

First(Y) = {implements, ε }

First(I) = {id}

First(J) = {',', ε }

Follow(C) = {$}

Follow(P) = {final, class}

Follow(F) = {class}

Follow(X) = {implements, $

Follow(Y) = {$}

Follow(I) = {$}

Follow(J) = {$}

# Construction of a 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 $\epsilon$ is in FIRST($\alpha$), then for each terminal $b$ in FOLLOW($A$), add $A \to \alpha$ to $M[A, b]$. If $\epsilon$ is in FIRST($\alpha$) and \$ is in FOLLOW($A$), add $A \to \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

TABLE: FIRST & FOLLOW

| Production | Symbol | FOLLOW |
|---|---|---|
| E -> TE' | {(, id} | {$, )} |
| E' -> +TE'\|ε | {+, ε} | {$, )} |
| T -> FT' | {(, id} | {+, $, )} |
| T' -> *FT'\|ε | {*, ε} | {+, $, )} |
| F -> (E)\|id | {(, id} | {*, +, $, )} |

TABLE: PARSING TABLE

| Non Terminal | INPUT SYMBOLS | | | | | |
|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ |
| E | E -> TE' | | | E -> TE' | | |
| E' | | E'-> +TE' | | | E' -> ε | E' -> ε |
| T | T -> FT' | | | T -> FT' | | |
| T' | | T' -> ε | T' -> *FT' | | T' -> ε | T' -> ε |
| F | F -> id | | | F -> (E) | | |

TABLE:
FAST & FOLLOW

| SYMBOL | FIRST | FOLLOW |
|---|---|---|
| S → iEtSS' \| a | a , i | $ , e |
| S' → eS \|ε | e, ε | $ , e |
| E → b | b | t |

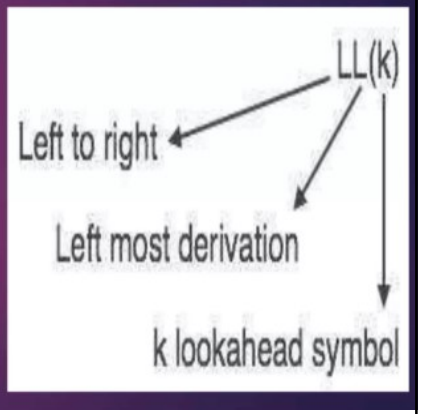
TABLE:
PARSING TABLE

| Non Terminal | INPUT SYMBOLS | | | | | |
|---|---|---|---|---|---|---|
| | a | b | e | i | t | $ |
| S | S→a | | | S→iEtSS' | | |
| S' | | | S' → eS<br>S'→ε | | | S'→ε |
| E | | E → b | | | | |

AMBIGUITY

- LL(1) grammars have distinct properties.
  - No ambiguous grammar or left recursive grammar can be LL(1).
- Thus , the given grammar is not LL(1).

- An LL parser is called an LL($k$) parser if it uses $k$ tokens of look ahead when parsing a sentence.
- LL grammars, particularly LL(1) grammars, as parsers are easy to construct, and many computer languages are designed to be LL(1) for this reason.
- The 1 stands for using **one** input symbol of look ahead at each step to make parsing action decision.




# Properties of LL(1)

- No ambiguous or left recursive grammar can be LL(1).
- Grammar G is LL(1) **iff** whenever $A \rightarrow \alpha / \beta$ are two distinct productions of G and:
  - For no terminal $a$ do both $\alpha$ and $\beta$ derive strings beginning with $a$.
    $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$
  - At most one of $\alpha$ and $\beta$ can derive the empty string.
  - If $\beta \Rightarrow \varepsilon$, the $\alpha$ does not derive any string beginning with a terminal in FOLLOW(A).
    $FIRST(\alpha FOLLOW(A)) \cap FIRST(\beta FOLLOW(A)) = \varnothing$

## Nonrecursive Predictive Parsing

- It is possible to build a nonrecursive predictive parser
- This is done by maintaining an explicit stack

2

## Table-driven Parsers

- The nonrecursive LL(1) parser looks up the production to apply by looking up a parsing table

3

## Table-driven Parsers

LL(1) table:

- One dimension for current non-terminal to expand
- One dimension for next token
- Table entry contains one production

Consider the expression grammar

| | | | |
|---|---|---|---|
| 1 | *E* | → | *T E′* |
| 2 | *E′* | → | + *T E′* |
| 3 | | \| | ε |
| 4 | *T* | → | *F T′* |
| 5 | *T′* | → | * *F T′* |
| 6 | | \| | ε |
| 7 | *F* | → | ( *E* ) |
| 8 | | \| | id |

## Predictive Parsing Table

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

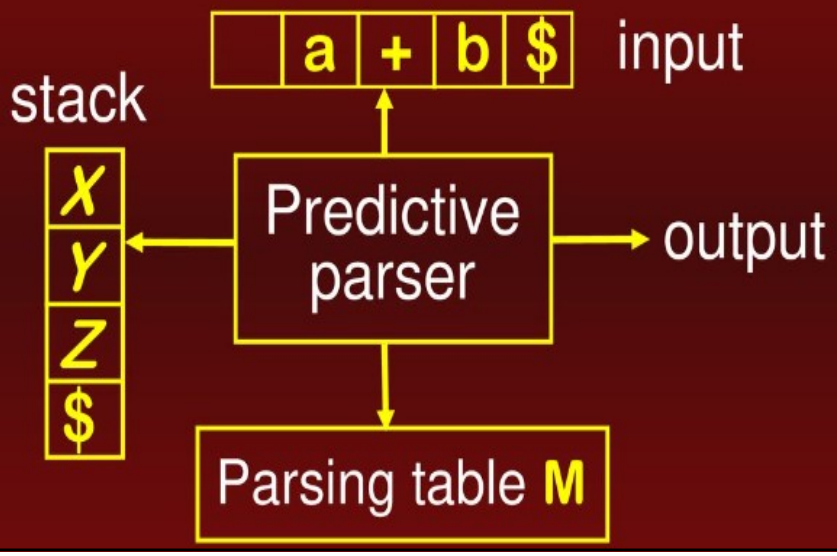
Columns for next token

Blank entries are errors

## Predictive Parsers

- The predictive parser uses an explicit stack to keep track of pending non-terminals
- It can thus be implemented without recursion.

## Predictive Parsers




## LL(1) Parsing Algorithm

- The *input buffer* contains the string to be parsed; **$** is the end-of-input marker
- The *stack* contains a sequence of grammar symbols
- Initially, the stack contains the start symbol of the grammar on the top of **$**.

The parser is controlled by a program that behaves as follows:

- The program considers $X$, the symbol on top of the stack, and **a**, the current input symbol.
- These two symbols, $X$ and **a** determine the action of the parser.
- There are *three* possibilities.

1. $X = a = \$$, the parser halts and annouces successful completion.
2. $X = a \neq \$$ the parser pops $X$ off the stack and advances input pointer to next input symbol
3. If $X$ is a nonterminal, the program consults entry **M**[$X$,**a**] of parsing table **M**.

If the entry is a production
$M[X,a] = \{X \rightarrow UVW\}$
the parser replaces $X$ on top of the stack by $WVU$ (with $U$ on top).

If $M[X,a]$ =**error**, the parser calls an error recovery routine

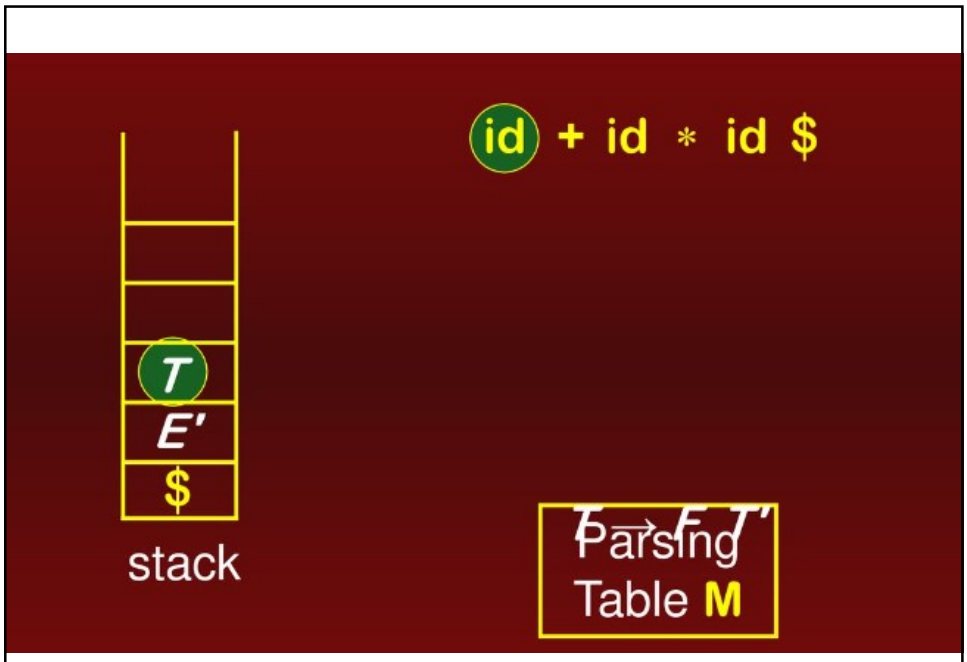
Example:

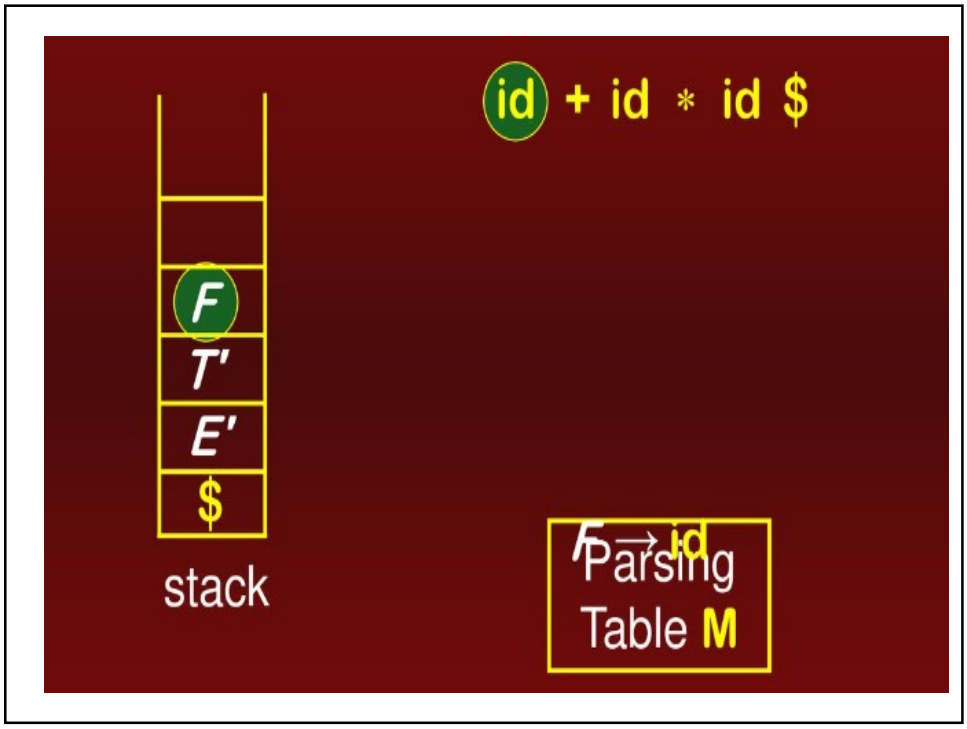
Let's parse the input string

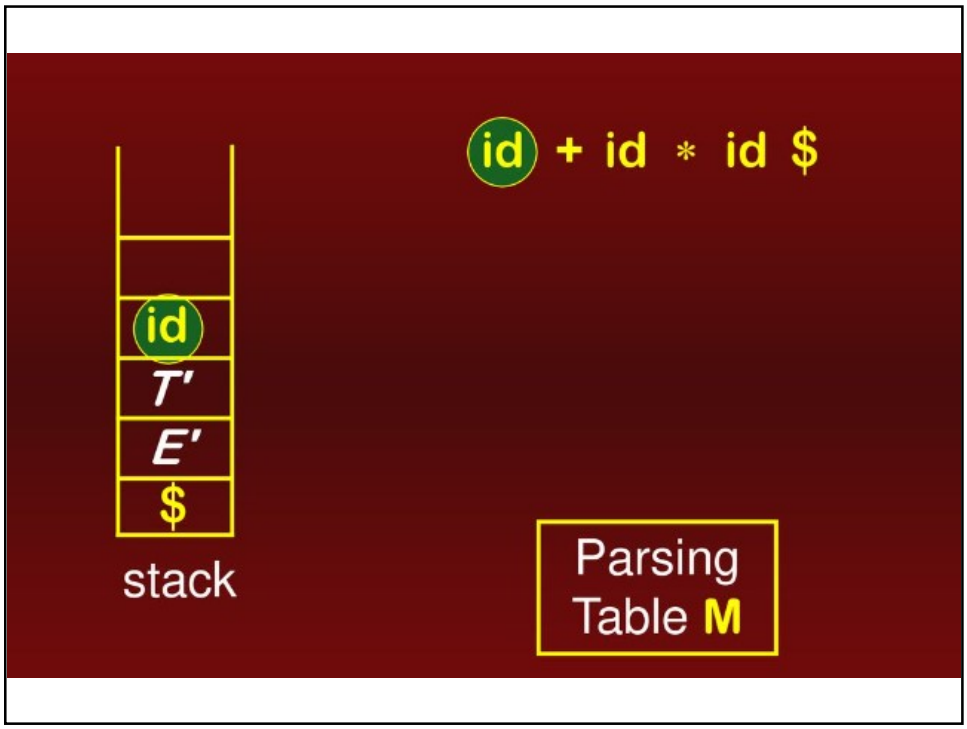
**id+id*id**

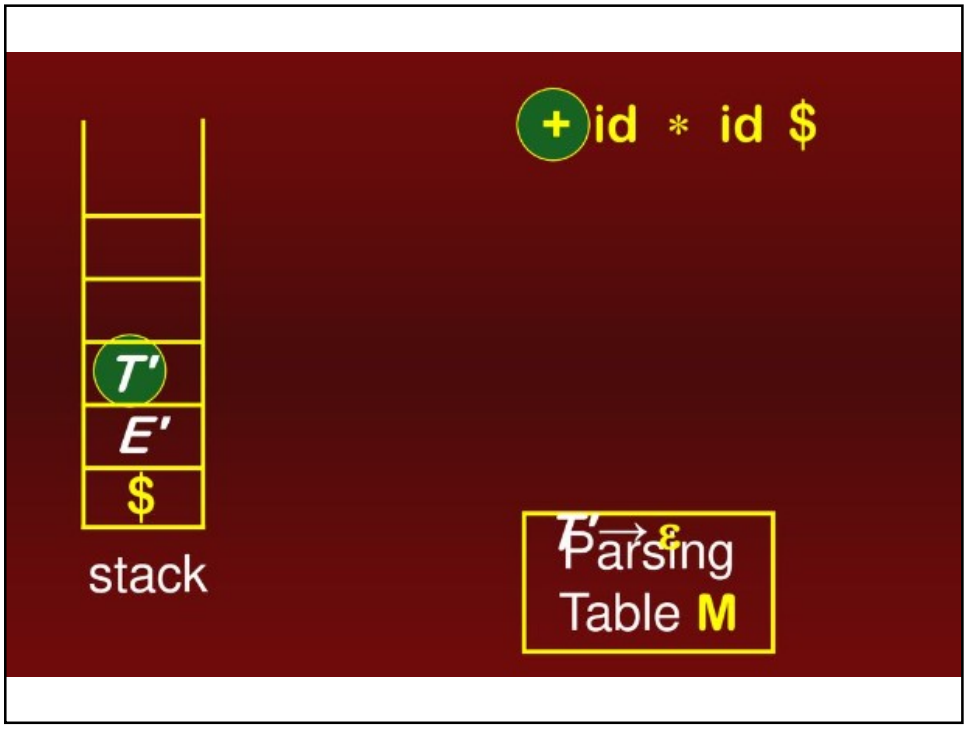
using the nonrecursive LL(1) parser

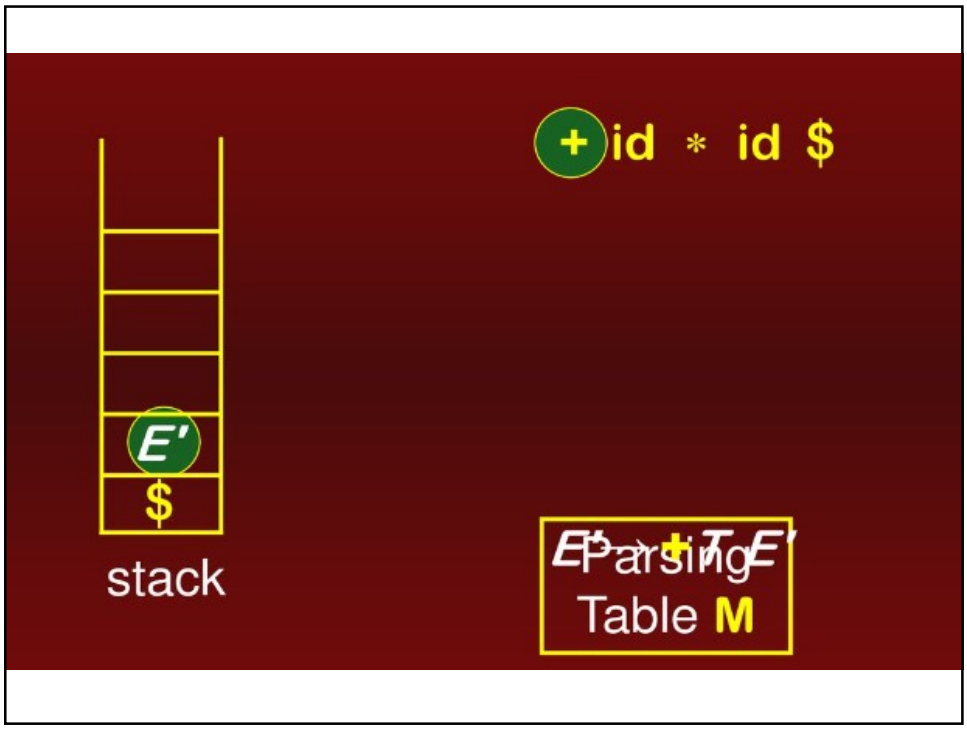

id
+
*
(
)
$
E
E → TE'
E → TE'
E'
E' → +TE'
E' → ε
E' → ε
T
T → FT'
T → FT'
T'
T' → ε
T → *FT'
T' → ε
T' → ε
F
F → id
F → (E)
id + id * id $
E
$
stack
E → TE'
Parsing Table M

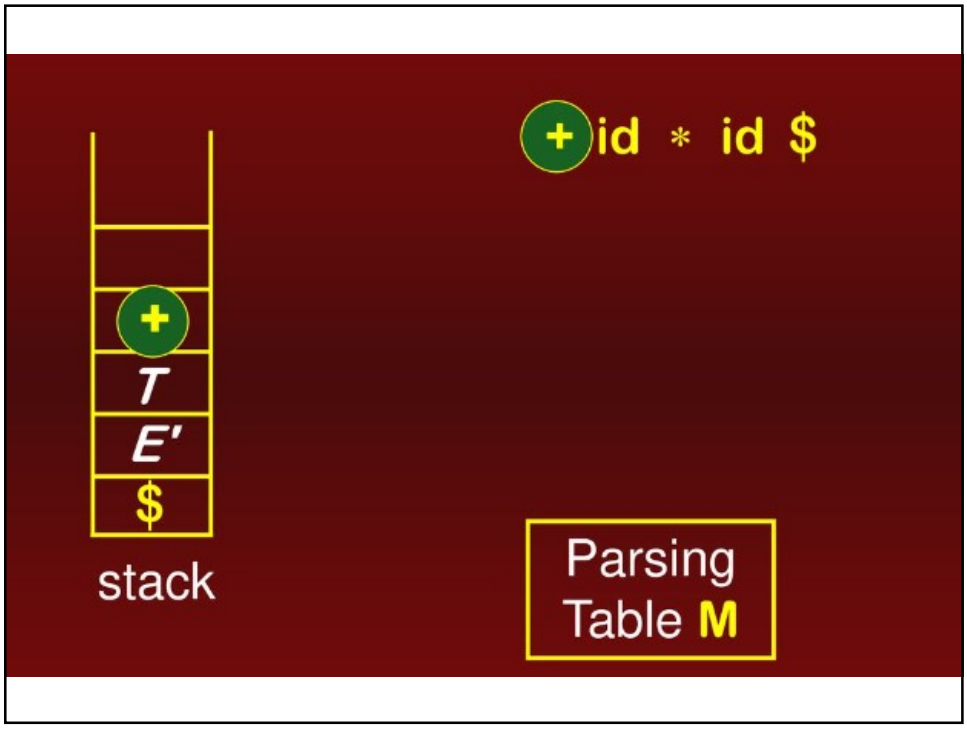

id + id * id $
T
E'
$
stack
T → FT'
Parsing Table M


id + id * id $
F
T'
E'
$
stack
F → id
Parsing
Table M

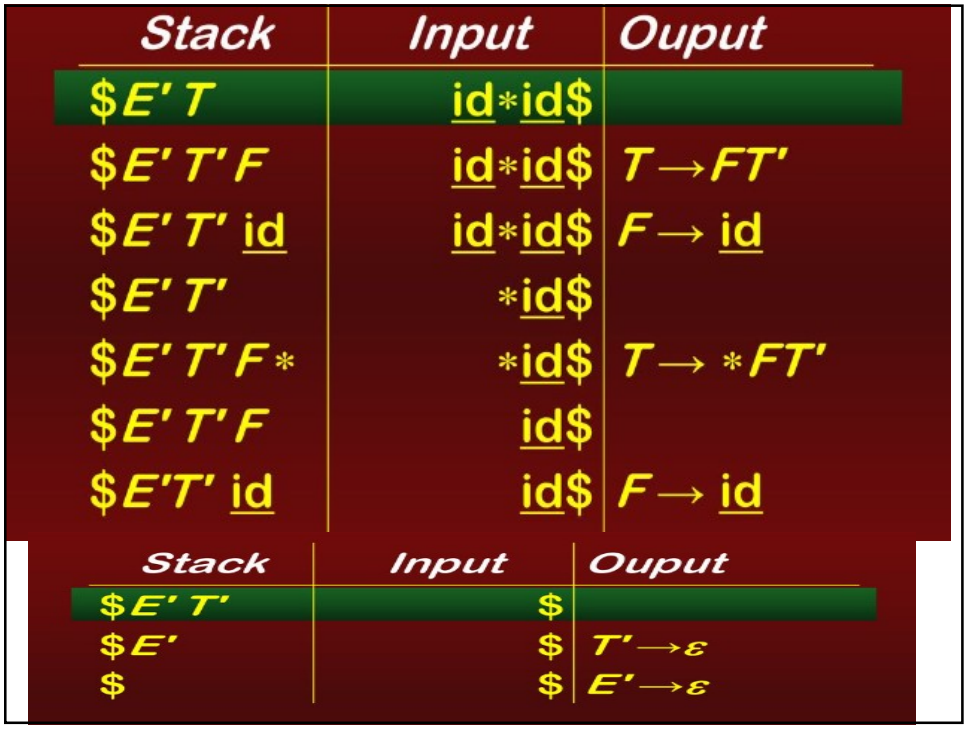

id + id * id $
id
T'
E'
$
stack
Parsing
Table M


+ id * id $
T'
E'
$
stack
T' → ε
Parsing
Table M


+ id * id $
E'
$
stack
E' → + T E'
Parsing
Table M




| Stack | Input | Ouput |
|---|---|---|
| $E | id+id*id$ | |
| $E' T | id+id*id$ | E → TE' |
| $E' T' F | id+id*id$ | T → FT' |
| $E'T' id | id+id*id$ | F → id |
| $E' T' | +id*id$ | |
| $E' | +id*id$ | T' → ε |
| $E' T+ | +id*id$ | E' → +TE' |



| Stack | Input | Ouput |
|---|---|---|
| $E' T | id*id$ | |
| $E' T' F | id*id$ | $T \rightarrow FT'$ |
| $E' T' id | id*id$ | $F \rightarrow id$ |
| $E' T' | *id$ | |
| $E' T' F* | *id$ | $T \rightarrow *FT'$ |
| $E' T' F | id$ | |
| $E'T' id | id$ | $F \rightarrow id$ |

| Stack | Input | Ouput |
|---|---|---|
| $E' T' | $ | |
| $E' | $ | $T' \rightarrow \varepsilon$ |
| $ | $ | $E' \rightarrow \varepsilon$ |

# Error Recovery in Predictive Parsing

- An error is detected during the predictive parsing when the terminal on top of the stack does not match the next input symbol, or when nonterminal A on top of the stack, a is the next input symbol, and parsing table entry M[A,a] is empty.
  - The parser design should be able to provide an error message (an error message which depicts as much possible information as it can ).
  - It should be recovering from that error case, and it should be able to continue parsing with the rest of the input.

## 4 Types of error recovery strategies

1. Panic-Mode Recovery
2. Phrase-Level Recovery
3. Error Productions
4. Global Correction

## Panic mode error recovery

- Panic-mode error recovery says that all the input symbols are skipped until a synchronizing token is found from the string.
- **What is the synchronizing token?**
- The terminal symbols which lie in follow set of non-terminal they can be used as a synchronizing token set for that non-terminal

## How to select synchronizing set?

- Place all symbols in FOLLOW(A) into the synchronizing set for non terminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it likely that parsing can continue.
- We might add keywords that begins statements to the synchronizing sets for the non terminals generating expressions.

- If a non terminal can generate the empty string, then the production deriving ε can be used as a default. This may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of non terminals that have to be considered during error recovery.
- If a terminal on top of stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted.

# Example: error recovery

"synch" indicating synchronizing tokens obtained from FOLLOW set of the nonterminal in question.

If the parser looks up entry M[A,a] and finds that it is blank, the input symbol a is skipped.

If the entry is synch, the the nonterminal on top of the stack is popped.

If a token on top of the stack does not match the input symbol, then we pop the token from the stack.

$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\}$.

$\text{FIRST}(E') = \{+, \epsilon\}$

$\text{FIRST}(T') = \{*, \epsilon\}$

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

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

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

| NONTER-MINAL | INPUT SYMBOL | | | | | |
|---|---|---|---|---|---|---|
| | **id** | + | * | ( | ) | $ |
| $E$ | $E \to TE'$ | | | $E \to TE'$ | synch | synch |
| $E'$ | | $E' \to +TE'$ | | | $E' \to \epsilon$ | $E' \to \epsilon$ |
| $T$ | $T \to FT'$ | synch | | $T \to FT'$ | synch | synch |
| $T'$ | | $T' \to \epsilon$ | $T' \to *FT'$ | | $T' \to \epsilon$ | $T' \to \epsilon$ |
| $F$ | $F \to \textbf{id}$ | synch | synch | $F \to (E)$ | synch | synch |

**Fig. 4.18.** Synchronizing tokens added to parsing table of Fig. 4.15.

# Example: error recovery (II)

| STACK | INPUT | REMARK |
|---|---|---|
| $E | ) id * + id $ | error, skip ) |
| $E | id * + id $ | id is in FIRST($E$) |
| $E'T | id * + id $ | |
| $E'T'F | id * + id $ | |
| $E'T'id | id * + id $ | |
| $E'T' | * + id $ | |
| $E'T'F* | * + id $ | |
| $E'T'F | + id $ | error, $M[F, +] =$ synch |
| $E'T' | + id $ | $F$ has been popped |
| $E' | + id $ | |
| $E'T+ | + id $ | |
| $E'T | id $ | |
| $E'T'F | id $ | |
| $E'T'id | id $ | |
| $E'T' | $ | |
| $E' | $ | |
| $ | $ | |

**Fig. 4.19.** Parsing and error recovery moves made by predictive parser.

# Example 2

S-› AbS|E|ε
A-› a|cAd

Follow(S)={$}
Follow(A)={b,d}

| | a | b | c | d | e | $ |
|---|---|---|---|---|---|---|
| S | S->Abs | sync | S->Abs | sync | S->e | S->ε |
| A | A->A | sync | A->cAd | sync | sync | sync |

| stack | input | output |
|---|---|---|
| $S | aab$ | S --> AbS |
| $SbA | aab$ | A --> a |
| $Sba | aab$ | |
| $Sb | ab$ | Error: missing b, inserted |
| $S | ab$ | S --> AbS |
| $SbA | ab$ | A --> a |
| $Sba | ab$ | |
| $Sb | b$ | |
| $S | $ | S --> ε |
| $ | $ | accept |

| stack | input | output |
|---|---|---|
| $S | ceadb$ | S --> AbS |
| $SbA | ceadb$ | A --> cAd |
| $SbdAc | ceadb$ | |
| $SbdA | eadb$ | Error: unexpected |
| e | | (illegal A) |
| | (Remove all input tokens until first b or d ,pop A) | |
| $Sbd | db$ | |
| $Sb | b$ | |
| $S | $ | S --> ε |
| $ | $ | accept |

# Phrase-level recovery

- Phrase level recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines.
- At each unfilled entry in the parsing table, it is filled by a pointer to a special error routine that will take care of that error case specifically.
- These error routines can be of different types like :
  - change, insert, or delete input symbols.
  - issue appropriate error messages.
  - pop items from the stack.
- We should be careful while we design these error routines because we may put the parser into an infinite loop

# Error productions

- If a user has knowledge of common errors that can be encountered then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.
- If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.

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

Grammar: S-> A

A-> aA | bA | a | b

B-> cd

The input string is not obtainable by the above grammar, so we need to add Augmented Grammar.

Grammar: E->SB // AUGMENT THE GRAMMAR

S-> A

A-> aA| bA | a | b

B-> cd

Now, string abcd is possible to obtain.

# Global corrections

- The parser examines the whole program and tries to find out the closest match for it which is error-free.
- The closest match program has less number of insertions, deletions, and changes of tokens to recover from erroneous input.
- Due to high time and space complexity, this method is not implemented practically.

**1. Universal parsers:-**

- These parsers perform parsing. Cocke - Kalami - Younger (CKY) algorithm. It uses the Chomsky Normal Form (CNF) of the CFG.
- It is in efficient and not used in Commercial Compilers.

**2. Top down parsers: -**

- These parsers build parse trees starting from the root node and work up to the Leaves.
- These parsers Satisfy LL grammars.
- These Scanners scan from left to right and use left-most derivation.
- These parsers are Categorized into
  - back track parsing.
  - Predictive parsing

- In back track parsing, if a sequence of erroneous expansions are made and a mismatch is discovered, the input pointer rolled back to the initial position and all effects of parsing are undone.
- In predictive parsing, using current non terminal to be expanded and the next input symbol, the parser makes a decision of which production needs to be expanded.
- **Two ways of implementing predictive parsers:**
- Recursive descent parsing have Procedure for Parsing each non-terminal using the recursive nature of The procedures.
- Table driven parsing. determines the next production to be applied by using a parsing table. Indexed by the Current non-terminal and the next input symbol.

**3. Bottom up Parsers:**

- There Parsers build parse trees starting from the leaves and work up to the root node.
- They satisfy LR grammars.
- They scan from left to right and use right most derivation.
- Here the input is reduced to the start symbol.
- These parsers are implemented using stacks.
- Here the next input symbol is shifted onto the stack
- Then the top few elements are popped out of the stack if a match occurs to right side of the production. It is then reduced to the left side of the production.
- Bottom up parsing is also called as shift reduce parsing.
- **These passers are categorized into**
  - Operator precedence parsers
  - LR parsers

- In operator precedence parsers, the choice of whether to shift the next symbol. Or reduce the processed input is decided using a precedence relation table.
- In LR Parsers, passing table derived out of grammar is used for deciding whether to shift the next input Symbol or reduce the processed input.

  These parsers are categorized as
  - Simple LR (SLR)
  - Canonical LR (CLR) and
  - Look Ahead LR (LALR) based on the way the parser tables are derived.