CSE 303 (Compilers)

Syntax Analysis

Tasmiah Tamzid Anannya

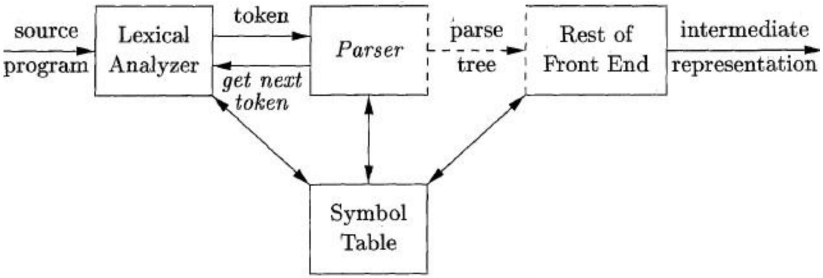
Lecturer

Department of Computer Science and Engineering
Military Institute of Science and Technology (MIST)
Mirpur Cantonment, Dhaka-1216, Bangladesh

Courtesy: Dr Mohammad Masroor Ali, Professor, BUET

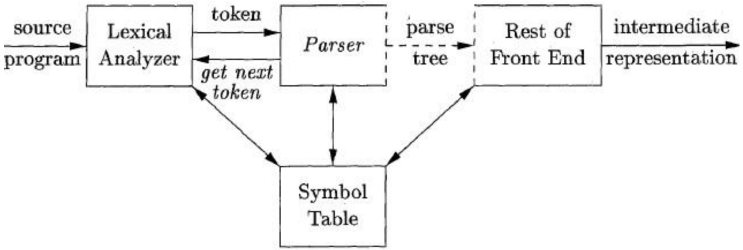
The Role of the Parser

- In our compiler model, the parser obtains a string of tokens from the lexical analyzer.
- It then verifies that the string of token names can be generated by the grammar for the source language.



Position of parser in compiler model

The Role of the Parser — *continued*



Position of parser in compiler model

- 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.

The Role of the Parser — *continued*

- There are three general types of parsers for grammars:
  - universal,
  - top-down, and
  - bottom-up.
- Universal parsing methods can parse any grammar.

- These general methods are, however, too inefficient to use in production compilers.

The Role of the Parser — *continued*

- 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.

Representative Grammars

- Some of the grammars that will be examined are presented here for ease of reference.
- Constructs that begin with keywords like **while** or **int**, are relatively easy to parse.
- The keyword guides the choice of the grammar production that must be applied to match the input.
- We therefore concentrate on expressions, which present more of challenge, because of the associativity and precedence of operators.

Representative Grammars

- Some of the grammars that will be examined are presented here for ease of reference.
- Constructs that begin with keywords like **while** or **int**, are relatively easy to parse.
- The keyword guides the choice of the grammar production that must be applied to match the input.
- We therefore concentrate on expressions, which present more of challenge, because of the associativity and precedence of operators.

Representative Grammars — *continued*

- Associativity and precedence are captured in the following grammar.
- E represents expressions consisting of terms separated by $+$ signs.
- T represents terms consisting of factors separated by $*$ signs.
- F represents factors that can be either parenthesized expressions or identifiers:

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

Representative Grammars — *continued*

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

- The above grammar belongs to the class of *LR* grammars that are suitable for bottom-up parsing.
- This grammar can be adapted to handle additional operators and additional levels of precedence.
- However, it cannot be used for top-down parsing because it is left recursive.

Representative Grammars — *continued*

- The following non-left-recursive variant of the expression grammar will be used for top-down parsing:

$$\begin{aligned} E &\rightarrow TE^{'} \\ E^{'} &\rightarrow +TE^{'} \,|\, s \\ T &\rightarrow FT^{'} \\ T' &\rightarrow *FT^{'} \,|\, s \\ F &\rightarrow (\, E\,) \,|\, \textbf{id} \end{aligned}$$

Representative Grammars — *continued*

- The following grammar treats + and $*$ alike.

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

- So it is useful for illustrating techniques for handling ambiguities during parsing.
- Here, E represents expressions of all types.
- This grammar permits more than one parse tree for expressions like $a + b * c$.

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.

Syntax Error Handling — *continued*

- Our civilization would be radically different if spoken languages had the same requirements for syntactic accuracy as computer languages.
- 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 — *continued*

Common programming errors can occur at many different levels.

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.

Syntax Error Handling — *continued*

Common programming errors can occur at many different levels.

Syntactic errors include misplaced semicolons or extra or missing braces, that is, "{" or "}".

As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error.

However, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code.

Syntax Error Handling — *continued*

Common programming errors can occur at many different levels.

Semantic errors include type mismatches between operators and operands.

An example is a `return` statement in a Java method with result type `void`.

Syntax Error Handling — *continued*

Common programming errors can occur at many different levels.

Logical errors can be anything from incorrect reasoning on the part of the programmer.

Syntax Error Handling — *continued*

- The precision of parsing methods allows syntactic errors to be detected very efficiently.
- Several parsing methods, such as the LL and LR methods, detect an error as soon as possible.
- That is, when the stream of tokens from the lexical analyzer cannot be parsed further according to the grammar for the language.
- More precisely, they have the viable-prefix property, meaning that they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.

Syntax Error Handling — *continued*

- 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.

Writing a Grammar

- Grammars are capable of describing most, but not all, of the syntax of programming languages.
- For instance, the requirement that identifiers be declared before they are used, cannot be described by a context-free grammar.
- Therefore, the sequences of tokens accepted by a parser form a superset of the programming language.
- Subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

Writing a Grammar

- Grammars are capable of describing most, but not all, of the syntax of programming languages.
- For instance, the requirement that identifiers be declared before they are used, cannot be described by a context-free grammar.
- Therefore, the sequences of tokens accepted by a parser form a superset of the programming language.
- Subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

Elimination of Left Recursion

- A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \overset{+}{\Longrightarrow} Aa$ for some string a.
- Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed.
- In simple left recursion there was one production of the form $A \rightarrow A\ \alpha$.
- Here we study the general case.

Elimination of Left Recursion — *continued*

- Left-recursive pair of productions $A \rightarrow A\,\alpha \,|\, \beta$ can be replaced by the non-left-recursive productions

  $$A \rightarrow \beta A'$$
  $$A' \rightarrow \alpha A' \,|\, \epsilon$$

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

Example

$$A \rightarrow A\alpha \mid \beta$$

to be replaced by

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

- Grammar for arithmetic expressions,

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

- Eliminating the immediate left recursions we obtain,

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

Elimination of Left Recursion — *continued*

- No matter how many A-productions there are, we can eliminate immediate left recursion from them.
- First, we group the A-productions as,

$$A \to A\alpha_1 \,|\, A\alpha_2 \,|\, A\alpha_3 \,|\, \ldots \,|\, A\alpha_m \,|\, \beta_1 \,|\, \beta_2 \,|\, \beta_3 \,|\, \ldots \,|\, \beta_n$$

where no β_i, begins with an A.

- Then, we replace the A-productions by,

$$\begin{aligned} A &\to \beta_1 A' \,|\, \beta_2 A' \,|\, \beta_3 A' \,|\, \ldots \,|\beta_n A' \\ A' &\to \alpha_1 A' \,|\, \alpha_2 A' \,|\, \alpha_3 A' \,|\, \ldots \,|\, \alpha_m A' \,|\, \epsilon \end{aligned}$$

- It does not eliminate left recursion involving derivations of two or more steps.

Elimination of Left Recursion — *continued*

- It does not eliminate left recursion involving derivations of two or more steps.
- Consider the grammar,

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \epsilon \end{aligned}$$

- The nonterminal S is left-recursive because $S \Rightarrow Aa \Rightarrow Sda$, but it is not immediately left recursive.

Algorithm

Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.
OUTPUT: An equivalent grammar with no left recursion.
METHOD: Apply the algorithm to G. Note that the resulting non-left-recursive grammar may have ϵ-productions.

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

Algorithm

Grammar with cycles: Grammar where derivations of the form

$A \overset{+}{\Longrightarrow} A$ occurs.

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

- In the first iteration for $i = 1$, the outer `for`-loop of lines (2) through (7) eliminates any immediate left recursion among A_1-productions.

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

- Any remaining A_1 productions of the form $A_1 \to A_l\alpha$ must therefore have $l > 1$.

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

- After the $i - 1$st iteration of the outer `for`- loop, all nonterminals A_k, where $k < i$, are “cleaned”.
- That is, any production $A_k \rightarrow A_l\,\alpha$, must have $l > k$.

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

- As a result, on the ith iteration, the inner loop of lines (3) through (5) progressively raises the lower limit in any production $A_i \rightarrow A_m \alpha$, until we have $m \geq i$.

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

- Then, eliminating immediate left recursion for the A_i productions at line (6) forces m to be greater than i.

Example

Input Grammar G with no cycles or s-productions.

- We apply the procedure to grammar,

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

- Technically, the algorithm is not guaranteed to work, because of the ϵ -production.
- But in this case the production $A \rightarrow \epsilon$ turns out to be harmless.

Example — *continued*

Left-Recursive Grammar

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \epsilon \end{aligned}$$

- We order the nonterminals S, A.
- $A_1 = S$, $A_2 = A$

Example — *continued*

Left-Recursive Grammar

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \epsilon \end{aligned}$$

- We order the nonterminals S, A.
- $A_1 = S$, $A_2 = A$

Example — *continued*

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

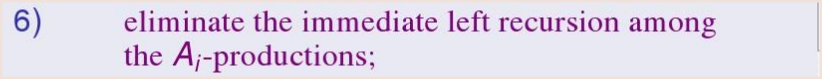
- $i = 1$, $A_1 = S$
- $j = 1$ to $j = 1 - 1 = 0$, the loop is *not* entered

Example — *continued*

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

- $i = 1$, $A_i = A_1 = S$
- $j = 1$ to $j = 1 - 1 = 0$, the loop is *not* entered

Example — *continued*



6) eliminate the immediate left recursion among the A_i-productions;

Left-Recursive Grammar

$$
\begin{array}{lcl}
S & \rightarrow & Aa \mid b \\
A & \rightarrow & Ac \mid Sd \mid \epsilon
\end{array}
$$

- There is no immediate left recursion among the *S*-productions, so nothing happens for the case $i = 1$. (A_i=$A_1 = S$)

Example — *continued*

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

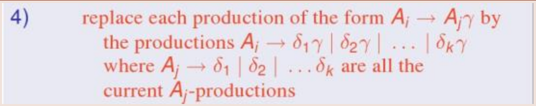
- $i = 2$, $A_i = A_2 = A$
- $j = 1$ to $j = 2 - 1 = 1$, the loop is entered

Example — *continued*

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

- $i = 2$, $A_i = A_2 = A$
- $j = 1$ to $j = 2 - 1 = 1$, the loop is entered

Example — *continued*



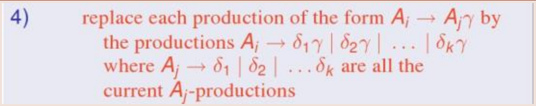
4) replace each production of the form $A_i \to A_j\gamma$ by the productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \to \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-Recursive Grammar

$$\begin{array}{lcl} S & \to & Aa \mid b \\ A & \to & Ac \mid Sd \mid \epsilon \end{array}$$

- $i = 2$, $A_i = A_2 = A$, $j = 1$, $A_j = A_1 = S$
- We need to
  - put productions of the form $S \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
  - in productions of the form $A \to S\gamma$
- Production(s) with S at the left-hand-side, $S \to Aa \mid b$
- Productions(s) with A at the left side and right side beginning with S is (are), $A \to Sd$

Example — *continued*

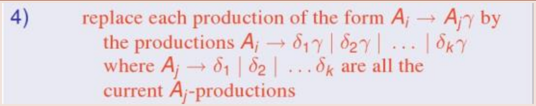

4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-Recursive Grammar

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

- $i = 2$, $A_i = A_2 = A$, $j = 1$, $A_j = A_1 = S$
- We need to
  - put productions of the form $S \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
  - in productions of the form $A \rightarrow S\gamma$
- Production(s) with S at the left-hand-side, $S \rightarrow Aa \mid b$
- Productions(s) with A at the left side and right side beginning with S is (are), $A \rightarrow Sd$

Example — *continued*

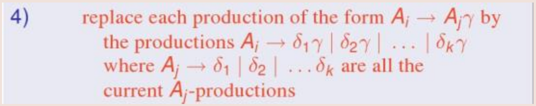

4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-Recursive Grammar

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \epsilon \end{aligned}$$

- $i = 2$, $A_i = A_2 = A$, $j = 1$, $A_j = A_1 = S$
- We need to
  - put productions of the form $S \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
  - in productions of the form $A \rightarrow S\gamma$
- Production(s) with S at the left-hand-side, $S \rightarrow Aa \mid b$
- Productions(s) with A at the left side and right side beginning with S is (are), $A \rightarrow Sd$

Example — *continued*



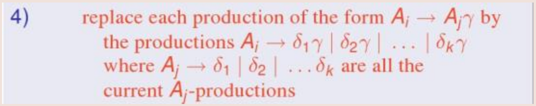
4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-Recursive Grammar

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \epsilon \end{aligned}$$

- $i = 2$, $A_i = A_2 = A$, $j = 1$, $A_j = A_1 = S$
- We need to
  - put productions of the form $S \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
  - in productions of the form $A \rightarrow S\gamma$
- Production(s) with S at the left-hand-side, $S \rightarrow Aa \mid b$
- Productions(s) with A at the left side and right side beginning with S is (are), $A \rightarrow Sd$

Example — *continued*



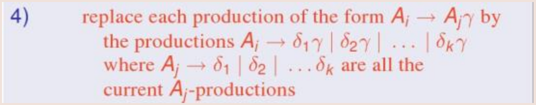
4) replace each production of the form $A_i \to A_j\gamma$ by the productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \to \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-Recursive Grammar

$$\begin{aligned} S &\to Aa \mid b \\ A &\to Ac \mid Sd \mid \epsilon \end{aligned}$$

- $i = 2, A_2 = A, j = 1, A_1 = S$
- We need to
  - put productions of the form $S \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
  - in productions of the form $A \to S\gamma$
- Production(s) with S at the left-hand-side, $S \to Aa \mid b$
- Productions(s) with A at the left side and right side beginning with S is (are), $A \to Sd$

Example — *continued*



4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

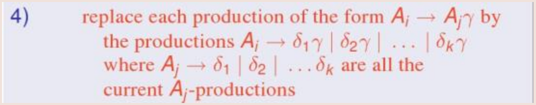
Left-Recursive Grammar

$$
\begin{aligned}
S &\rightarrow Aa \mid b \\
A &\rightarrow Ac \mid Sd \mid \epsilon
\end{aligned}
$$

- $S \rightarrow Aa \mid b$ to be put in A, $A \rightarrow Sd$
- We substitute $S \rightarrow Aa \mid b$ in $A \rightarrow Sd$ to get the following A-productions,

$$A \rightarrow Aad \mid bd$$

Example — *continued*



4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions

Left-Recursive Grammar

$$
\begin{aligned}
S &\rightarrow Aa \mid b \\
A &\rightarrow Ac \mid Sd \mid \epsilon
\end{aligned}
$$

- $S \rightarrow Aa \mid b$ to be put in A, $A \rightarrow Sd$
- We substitute $S \rightarrow Aa \mid b$ in $A \rightarrow Sd$ to get the following
  A-productions,

$$A \rightarrow Aad \mid bd$$

Example — *continued*

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

Example — *continued*

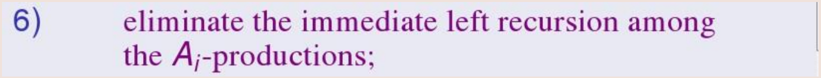
- All $A_i = A_2 = A$-productions together,

$$A \rightarrow \; Ac \mid Aad \mid bd \mid \epsilon$$

- Eliminating the immediate left recursion among the A-productions yields the following,

$$\begin{aligned} A &\rightarrow bdA^j \mid A^j \\ A^j &\rightarrow cA^j \mid adA^j \mid s \end{aligned}$$

Example — *continued*


6) eliminate the immediate left recursion among the A_i-productions;

- All $A_i = A_2 = A$-productions together,

$$A \longrightarrow Ac \mid Aad \mid bd \mid \epsilon$$

- Eliminating the immediate left recursion among the A-productions yields the following,

$$\begin{aligned} A &\longrightarrow bdA^{'} \mid A^{'} \\ A^{j} &\longrightarrow cA' \mid adA^{'} \mid \epsilon \end{aligned}$$

Example — *continued*

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

i has attained the value of $n = 2$ and the loops are no more entered.

Example — *continued*

Left-Recursive Grammar

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \epsilon \end{aligned}$$

- Put together we get the following non-left-recursive grammar,

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow bdA' \mid A' \\ A' &\rightarrow cA' \mid adA' \mid \epsilon \end{aligned}$$

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

Conceptual Technique Summary (Again)

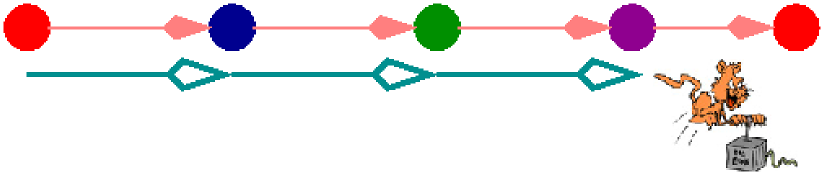
- Put some order in the nonterminals.
- Start by making first nonterminal productions left-recursion-free.
- Put the first nonterminal left-recursion-free productions into those of the second one.
- Now make the productions of second nonterminal left-recursion-free.
- Thus keep on growing the set of left-recursion-free productions.

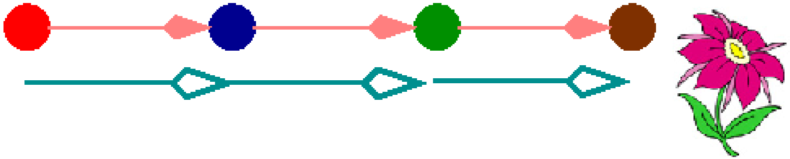
Left Factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- The basic idea is that sometimes it is not clear which of two alternative productions to use to expand a nonterminal A.
- We may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

Left Factoring — *continued*

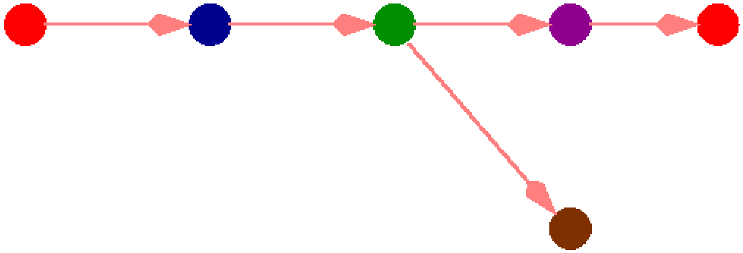
Road Direction: *Red* → *Blue* → *Green* → *Brown*





Left Factoring — *continued*

Defer the decision until we have seen enough of the input to make the right choice.



Left Factoring — *continued*

- We have the two productions,

$$\begin{aligned} stmt \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ |\ & \textbf{if}\ expr\ \textbf{then}\ stmt \end{aligned}$$

- On seeing the input token **if**, we cannot immediately tell which production to choose to expand *stmt*.

Left Factoring — *continued*

- $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions.
- The input begins with a nonempty string derived from α .
- We do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$.
- However, we may defer the decision by expanding A to $\alpha A'$.
- Then, after seeing the input derived from α we expand A' to β_1 or β_2.
- Left-factored, the original productions become,

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

Left Factoring Algorithm

INPUT. Grammar *G*.

OUTPUT An equivalent left-factored grammar.

Left Factoring Algorithm — *continued*

Method.

- For each nonterminal A find the longest prefix α common to two or more of its alternatives.
- If $\alpha \neq \epsilon$ (there is a nontrivial common prefix), replace all the A productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$ where γ represents all alternatives that do not begin with α by

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' = \beta_1 | \beta_2 | \beta_3 \cdots\cdots\cdots\cdots | \beta_n$$

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

Example

- The following grammar abstracts the dangling-else problem:

$$\begin{aligned} S &\rightarrow iEtS \mid iEtSeS \mid a \\ E &\rightarrow b \end{aligned}$$

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

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow eS \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

Example — *continued*

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow eS \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

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

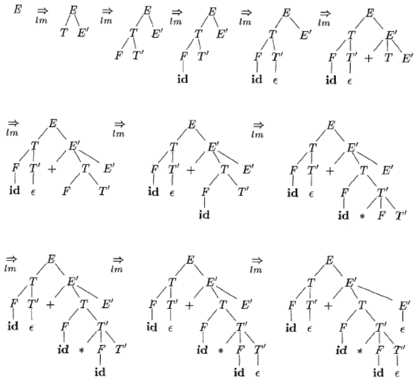
Top-Down Parsing

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

Example

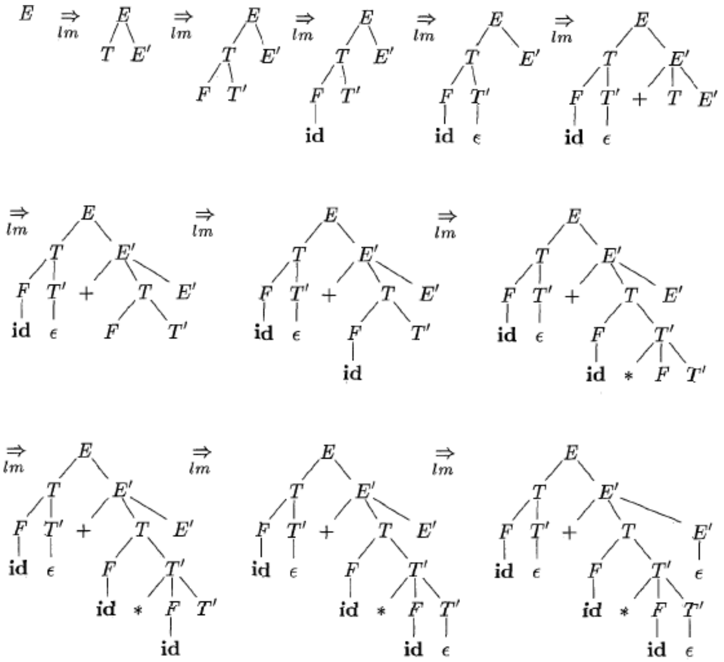
- The sequence of parse trees for the input **id** + **id** $*$**id** is a top-down parse according to grammar.

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



Top-down parse for **id** + **id** $*$ **id**

- This sequence of trees corresponds to a leftmost derivation of the input.

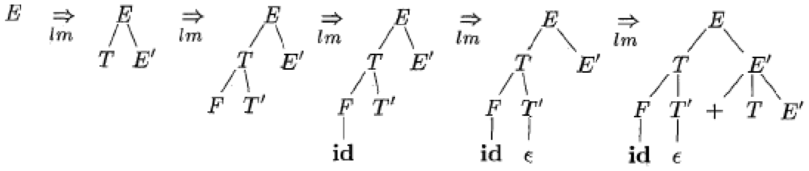


Top-down parse for **id** + **id** ∗ **id**

Top-Down Parsing — *continued*

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

Top-Down Parsing — *continued*



- Consider the top-down parse in figure.
- This constructs a tree with two nodes labeled E'.
- At the first E' node (in preorder), the production $E' \to +TE'$ is chosen.
- At the second E' node, the production $E' \to \epsilon$ is chosen.
- A predictive parser can choose between E'-productions by looking at the next input symbol.

Top-Down Parsing — *continued*

- The class of grammars for which we can construct predictive parsers looking *k* symbols ahead in the input is sometimes called the LL(*k*)class.
- We will discuss LL(1) parser.

FIRST and FOLLOW

- The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar *G*.
- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
- During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

FIRST and FOLLOW

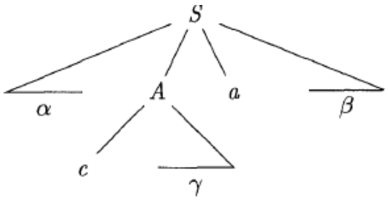
- The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G.
- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
- During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

FIRST and FOLLOW

- The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar *G*.
- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
- During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

FIRST and FOLLOW — *continued*

- Define FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α.
- If $\alpha \Rightarrow \epsilon$, then ϵ is also in FIRST(α).
- For example, in figure $A \Rightarrow c\gamma$, so c is in FIRST(A).



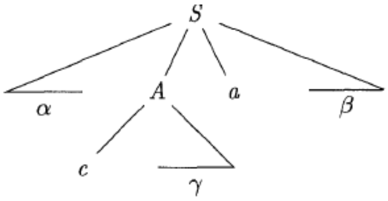
Terminal c is in FIRST(A) and a is in FOLLOW(A)

FIRST and FOLLOW — *continued*

- Let us see how FIRST can be used during predictive parsing.
- Consider two A-productions $A \rightarrow \alpha \mid \beta$, where FIRST($\alpha$) and FIRST($\beta$) are disjoint sets.
- We can then choose between these A-productions by looking at the next input symbol a, since a can be in at most one of FIRST(α) and FIRST(β), not both.
- For instance, if a is in FIRST(β) choose the production $A \rightarrow \beta$.

FIRST and FOLLOW — *continued*

- Define FOLLOW(*A*), nonterminal *A*, to be the set of terminals *a* that can appear immediately to the right of *A* in some sentential form.
- That is, the set of terminals *a* such that there exists a derivation of the form $S \Rightarrow \alpha A a \beta$, for some α and β.
- Note that there may have been symbols between *A* and *a*, at some time during the derivation, but if so, they derived ϵ and disappeared.



Terminal c is in FIRST(A) and a is in FOLLOW(A)

FIRST and FOLLOW — *continued*

- In addition, if A can be the rightmost symbol in some sentential form, then \$ is in FOLLOW(A).
- Recall that \$ is a special “endmarker” symbol that is assumed not to be a symbol of any grammar.

FIRST and FOLLOW — *continued*

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

1. For a production rule $X \rightarrow \in$, First(X) = { ∈ }

2. For any terminal symbol 'a', First(a) = { a }

3. For a production rule $X \rightarrow Y_1Y_2Y_3$,

- If $\epsilon \notin$ First(Y_1), then First(X) = First(Y_1)
- If $\epsilon \in$ First(Y_1), then First(X) = { First(Y_1) – ϵ } ∪ First(Y_2Y_3)
- Then, If $\epsilon \notin$ First(Y_2), then First(Y_2Y_3) = First(Y_2)
- If $\epsilon \in$ First(Y_2), then First(Y_2Y_3) = { First(Y_2) – ϵ } ∪ First(Y_3)
- Similarly, we can make expansion for any production rule

$X \rightarrow Y_1Y_2Y_3.....Y_n$.

FIRST and FOLLOW — *continued*

For a production rule $X \rightarrow Y_1Y_2Y_3$,

- If $\epsilon \notin$ First(Y_1), then First(X) = First(Y_1)
- If $\epsilon \in$ First(Y_1), then First(X) = { First(Y_1) – ϵ } ∪ First(Y_2Y_3)
- Then, If $\epsilon \notin$ First(Y_2), then First(Y_2Y_3) = First(Y_2)
- If $\epsilon \in$ First(Y_2), then First(Y_2Y_3) = { First(Y_2) – ϵ } ∪ First(Y_3)
- Similarly, we can make expansion for any production rule

$X \rightarrow Y_1Y_2Y_3.....Y_n$.

- For example, everything in FIRST(Y_1) is surely in FIRST(X).
- If Y_1, does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$) and so on.

FIRST and FOLLOW — *continued*

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

1. For the start symbol S, place $ in Follow(S).

2. For any production rule A → αB, Follow(B) = Follow(A)

3. For any production rule A → αBβ,

- If $\epsilon \notin$ First(β), then Follow(B) = First(β)
- If $\epsilon \in$ First(β), then Follow(B) = { First(β) – ϵ } ∪ Follow(A)

FIRST and FOLLOW — *continued*

- ϵ may appear in the FIRST function of a non-terminal.
- ϵ will never appear in the FOLLOW function of a non-terminal.
- Before calculating the FIRST and FOLLOW functions, eliminate Left Recursion from the grammar, if present.

Example

1. For a production rule $X \rightarrow \in$, $\text{First}(X) = \{ \in \}$
2. For any terminal symbol ‘a’, First(a) = { a }
3. For a production rule $X \rightarrow Y_1Y_2Y_3$,

- If $\epsilon \notin \text{First}(Y_1)$, then $\text{First}(X) = \text{First}(Y_1)$
- If $\epsilon \in \text{First}(Y_1)$, then $\text{First}(X) = \{ \text{First}(Y_1) - \epsilon \} \cup \text{First}(Y_2Y_3)$
- Then, If $\epsilon \notin \text{First}(Y_2)$, then $\text{First}(Y_2Y_3) = \text{First}(Y_2)$
- If $\epsilon \in \text{First}(Y_2)$, then $\text{First}(Y_2Y_3) = \{ \text{First}(Y_2) - \epsilon \} \cup \text{First}(Y_3)$
- Similarly, we can make expansion for any production rule $X \rightarrow Y_1Y_2Y_3.....Y_n$.

Grammar,

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow (E) \mid \textbf{id}$$

Then,

$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\}$$
$$\text{FIRST}(E') = \{+, \epsilon\}$$
$$\text{FIRST}(T') = \{*, \epsilon\}$$

Example

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).

Grammar,

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

Then,

FIRST(E) = FIRST(T) =FIRST(F) = $\{($, **id**$\}$

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

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

Example

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

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

Grammar,

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid s$$
$$F \rightarrow (E) \mid \textbf{id}$$

Then,

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

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

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

Example

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

- The reasoning is analogous to that for FIRST(E').

Grammar,

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

Then,

FIRST(E) = FIRST(T) = FIRST(F) = $\{(, \mathbf{id}\}$

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

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

Example — *continued*

- Grammar:

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

- Computation of FOLLOW:

FOLLOW(E)	FOLLOW(E^j)	FOLLOW(T)	FOLLOW(T^j)	FOLLOW(F)

Initially all sets are empty

Put \$ in FOLLOW(E) by rule (1) *(Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker)*

\$				

Example — *continued*

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

FIRST(E) = FIRST(T) = FIRST(F) = {(, **id**}, FIRST(E') = {+, ϵ}, FIRST(T') = {$*$, ϵ}

By rule (3) (If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for ϵ is placed in FOLLOW(B)) applied to,

$E \rightarrow TE'$: FIRST(E') except ϵ i.e. {+} are in FOLLOW(T)

$E' \rightarrow +TE'$: FIRST(E')except ϵ i.e. {+} are in FOLLOW(T)

$T \rightarrow FT'$: FIRST(T') except ϵ i.e. {$*$}are in FOLLOW(F)

$T \rightarrow *FT'$: FIRST(T') except ϵ i.e. {$*$}are in FOLLOW(F)

$F \rightarrow (E)$: FIRST()) i.e. {)} are in FOLLOW(E)

FOLLOW(E)	FOLLOW(E')	FOLLOW(T)	FOLLOW(T')	FOLLOW(F)
\$,)		+		$*$

Rule (2) is not applicable any more since it depends only on FIRST, which are now stable sets.

Example — *continued*

Application of rule (3) *(If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$ where FIRST(β) contains s (i.e., $\beta \overset{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

$$
\begin{array}{lll}
E \to TE' & T \to FT' & F \to (E) \mid \mathbf{id} \\
E' \to +TE' \mid \epsilon & T' \to *FT' \mid \epsilon &
\end{array}
$$

FOLLOW(*E*)	FOLLOW(*E'*)	FOLLOW(*T*)	FOLLOW(*T'*)	FOLLOW(*F*)
$,)		+		*

$E \to TE'$: Everything in FOLLOW(E) are in FOLLOW(E')

$,)	$,)	+		*

$E' \to +TE'$ (also ϵ is in FIRST(E')): Everything in FOLLOW(E') are in FOLLOW(T)

$,)	$,)	+, $,)		*

$T \to FT'$: Everything in FOLLOW(T) are in FOLLOW(T')

$,)	$,)	+, $,)	+, $,)	*

Example — *continued*

Application of rule (3) *(If there is a production $A \to \alpha B$, or a production $A \to \alpha B\beta$ where FIRST(β) contains s (i.e., $\beta \overset{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

$$\begin{array}{lll} E \to TE' & T \to FT' & F \to (E)\mid \textbf{id} \\ E' \to +TE' \mid \epsilon & T' \to *FT' \mid \epsilon & \end{array}$$

FOLLOW(E)	FOLLOW(E')	FOLLOW(T)	FOLLOW(T')	FOLLOW(F)
\$,)		+		$*$

$E \to TE'$: Everything in FOLLOW(E) are in FOLLOW(E')

\$,)	\$,)	+		$*$

$E' \to +TE'$ (also ϵ is in FIRST(E')): Everything in FOLLOW(E') are in FOLLOW(T)

\$,)	\$,)	+, \$,)		$*$

$T \to FT'$: Everything in FOLLOW(T) are in FOLLOW(T')

\$,)	\$,)	+, \$,)	+, \$,)	$*$

Example — *continued*

Application of rule (3) *(If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains s (i.e., $\beta \overset{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

$E \rightarrow TE'$ $\quad T \rightarrow FT'$ $\quad F \rightarrow (E) \mid \textbf{id}$

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

FOLLOW(E)	FOLLOW(E')	FOLLOW(T)	FOLLOW(T')	FOLLOW(F)
$,)		+		*

$E \rightarrow TE'$: Everything in FOLLOW(E) are in FOLLOW(E')

$,)	$,)	+		*

$E' \rightarrow +TE'$ (also ϵ is in FIRST(E')): Everything in FOLLOW(E') are in FOLLOW(T)

$,)	$,)	+, $,)		*

$T \rightarrow FT'$: Everything in FOLLOW(T) are in FOLLOW(T')

$,)	$,)	+, $,)	+, $,)	*

Example — *continued*

Application of rule (3) *(If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains s (i.e., $\beta \stackrel{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

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

FOLLOW(E)	FOLLOW(E')	FOLLOW(T)	FOLLOW(T')	FOLLOW(F)
$,)		+		*

$E \rightarrow TE'$: Everything in FOLLOW(E) are in FOLLOW(E')

$,)	$,)	+		*

$E' \rightarrow +TE'$ (also ϵ is in FIRST(E')): Everything in FOLLOW(E') are in FOLLOW(T)

$,)	$,)	+, $,)		*

$T \rightarrow FT'$: Everything in FOLLOW(T) are in FOLLOW(T')

$,)	$,)	+, $,)	+, $,)	*

Example — *continued*

Application of rule (3) *(If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains s (i.e., $\beta \overset{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

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

FOLLOW(E)	FOLLOW(E')	FOLLOW(T)	FOLLOW(T')	FOLLOW(F)
\$,)		+		$*$

$E \rightarrow TE'$: Everything in FOLLOW(E) are in FOLLOW(E')

\$,)	\$,)	+		$*$

$E' \rightarrow +TE'$ (also ϵ is in FIRST(E')): Everything in FOLLOW(E') are in FOLLOW(T)

\$,)	\$,)	+, \$,)		$*$

$T \rightarrow FT'$: Everything in FOLLOW(T) are in FOLLOW(T')

\$,)	\$,)	+, \$,)	+, \$,)	$*$

Example — *continued*

Application of rule (3) *(If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$ where FIRST(β) contains s (i.e., $\beta \overset{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

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

FOLLOW(E)	FOLLOW(E')	FOLLOW(T)	FOLLOW(T')	FOLLOW(F)
\$,)	\$,)	+, \$,)	+, \$,)	$*$

$T' \to *FT'$ (also s ϵFIRST(T')): Everything in FOLLOW(T') are in FOLLOW(F)

\$,)	\$,)	+, \$,)	+, \$,)	$*$,+, \$,)

We can try applying Rule (3) again, but will find that the sets have stabilized (nothing can be added to any FOLLOW set).

Practice problems- FIRST and FOLLOW

Problem-1 :

S-> ACB | CbB | Ba

A->da | BC

B-> g | ϵ

C-> h | ϵ

Problem-2 :

S-> (L) | a

L-> SL′

L′-> ,SL′ | ϵ

Problem-3:

S-> A

A-> aB | Ad

B-> b

C->g

HINT:
This grammar on prolem-3 is left recursive,
you must eliminate left recursion before finding
FIRST.

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.

Algorithm for Construction of a 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, 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).

Example

- For the expression grammar below,

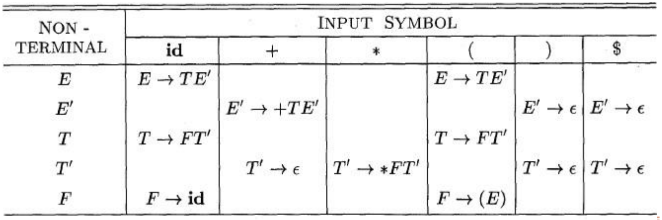
$$E \rightarrow TE^{'} \qquad T \rightarrow FT^{'} \qquad F \rightarrow (E) \mid \mathbf{id}$$
$$E^{'} \rightarrow +TE^{'} \mid \epsilon \qquad T^{'} \rightarrow *FT^{'} \mid \epsilon$$

FIRST(E) = FIRST(T) = FIRST(F) = $\{(, \mathbf{id}\}$

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

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

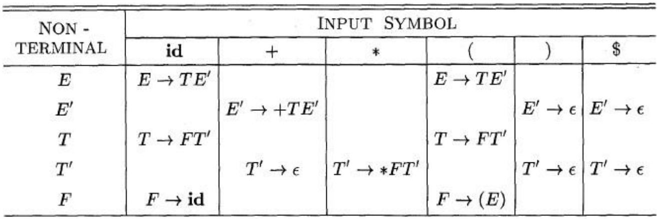
FOLLOW(E)	FOLLOW($E^{'}$)	FOLLOW(T)	FOLLOW($T^{'}$)	FOLLOW(F)
\$,)	\$,)	+, \$,)	+, \$,)	$*$, +, \$,)



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	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

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

1. For each terminal a in FIRST(α), 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.



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

- Consider production $E \to TE'$.
- Since

$$\text{FIRST}(TE') = \text{FIRST}(T) = \{(, \mathbf{id}\}$$

this production is added to $M[E, (\,]$ and $M[E, \mathbf{id}]$.

- Production $E' \to +TE'$ is added to $M[E', +]$ since FIRST($+TE'$) = {+}.

- Since FOLLOW(E') = {), \$}, production $E' \to \epsilon$ is added to $M[E',)]$ and $M[E', \$]$

Algorithm . . . Predictive Parsing Table — *continued*

- The aforementioned algorithm can be applied to any grammar *G* to produce a parsing table *M*.
- For every LL(1) grammar, each parsing-table entry uniquely identifies a production or signals an error.

Algorithm . . . Predictive Parsing Table — *continued*

- For some grammars, however, *M* may have some entries that are multiply defined.
- For example, if *G* is left-recursive or ambiguous, then *M* will have at least one multiply defined entry.
- Although left-recursion elimination and left factoring are easy to do, there are some grammars for which no amount of alteration will produce an LL(1) grammar.
- The language in the following example has no LL(1) grammar at all.

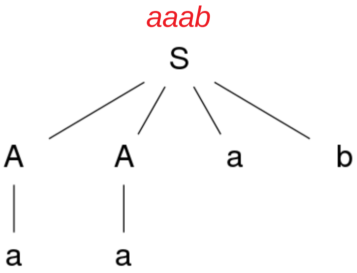
LL(1) Grammars — *continued*

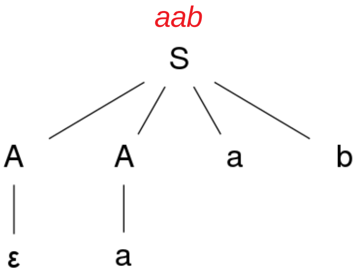
A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a . Meaning, FIRST(α) and FIRST(β) needs to be disjoint sets.
2. At most one of α and β can derive the empty string.
3. If $\beta \overset{*}{\Rightarrow} \epsilon$ then FIRST(α) and FOLLOW(A) needs to be disjoint. Likewise, $\alpha \overset{*}{\Rightarrow} \epsilon$, then FIRST(β) and FOLLOW(A) needs to be disjoint.

A Case of a non-LL(1) Grammar

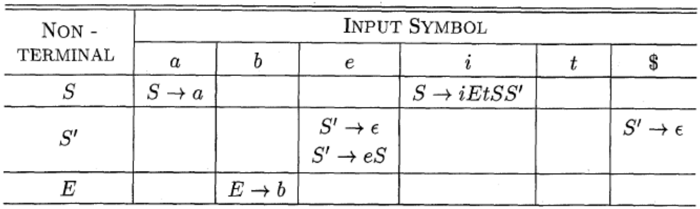
$$
\begin{aligned}
S &\rightarrow AAab \mid BbBa \\
A &\rightarrow a \mid \epsilon \\
B &\rightarrow b \mid \epsilon
\end{aligned}
$$





Example — *continued*

$$\begin{aligned} S &\rightarrow iEtSS^{'} \mid a \\ S^{'} &\rightarrow eS \mid \epsilon \\ E &\rightarrow b \end{aligned}$$



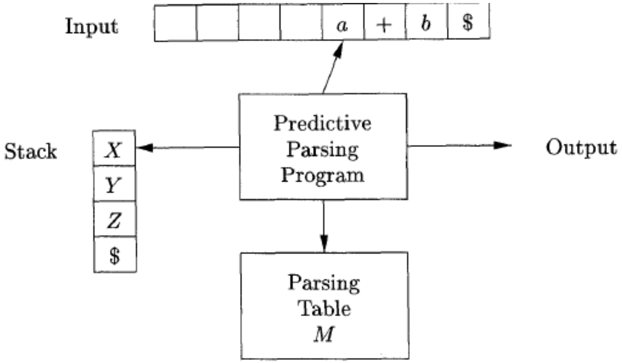
NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Nonrecursive 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

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

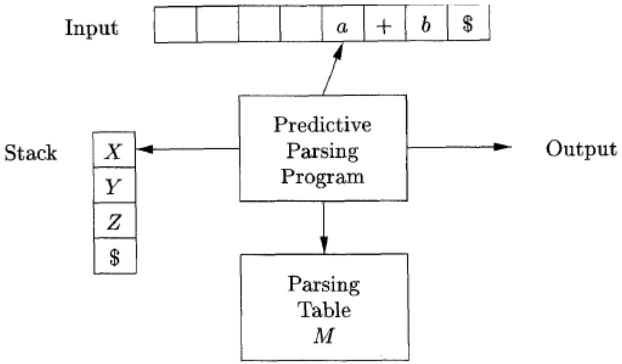
Nonrecursive Predictive Parsing



Model of a table-driven predictive parser

- The table-driven parser in figure has an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by algorithm, and an output stream.

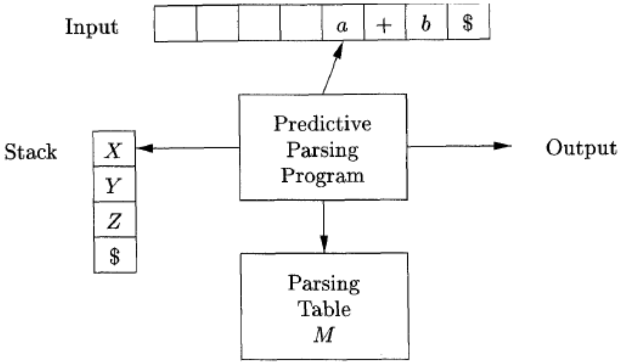
Nonrecursive Predictive Parsing



Model of a table-driven predictive parser

- 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 $.

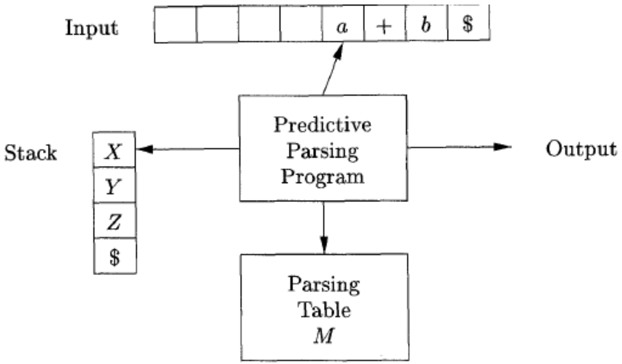
Nonrecursive Predictive Parsing



Model of a table-driven predictive parser

- The parser is controlled by a program that considers *X*, the symbol on top of the stack, and *a*, the current input symbol.

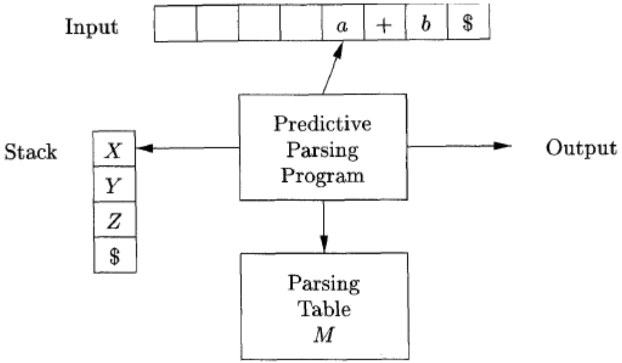
Nonrecursive Predictive Parsing



Model of a table-driven predictive parser

- If *X* is a nonterminal, the parser chooses an *X* -production by consulting entry *M*[*X*, *a*] of the parsing table *M*.
- Additional code could be executed here, for example, code to construct a node in a parse tree.

Nonrecursive Predictive Parsing



Model of a table-driven predictive parser

- Otherwise, it checks for a match between the terminal *X* and current input symbol *a*.

Nonrecursive Predictive Parsing — *continued*

```
set ip to point to the first symbol of $w$;
set $X$ to the top stack symbol;
while ( $X \neq \$$ ) { /* 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 ( $M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$ ) {
        output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$;
        pop the stack;
        push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with $Y_1$ on top;
    }
    set $X$ to the top stack symbol;
}
```

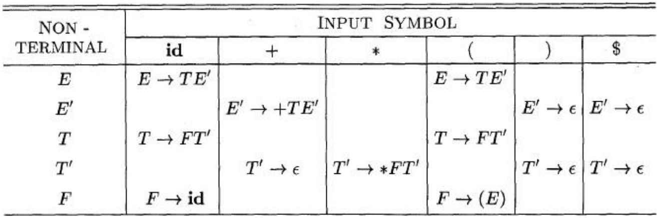
Predictive parsing algorithm

Example

- We consider grammar:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \,|\, s \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \,|\, s \\ F &\rightarrow (E) \,|\, \textbf{id} \end{aligned}$$

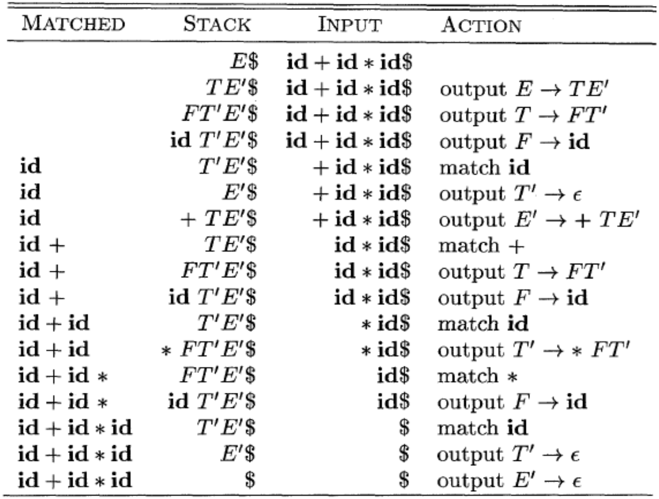
- We have already seen its parsing table.



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	$F \rightarrow \textbf{id}$			$F \rightarrow (E)$		

Example

- On input **id** + **id** * **id**, the nonrecursive predictive parser algorithm makes the sequence of moves,



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 $E' \to \epsilon$

Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$

Example

- These moves correspond to a leftmost derivation,

$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} \mathbf{id}T'E' \underset{lm}{\Rightarrow} \mathbf{id}E' \underset{lm}{\Rightarrow} \mathbf{id} + TE' \underset{lm}{\Rightarrow} \cdots$$

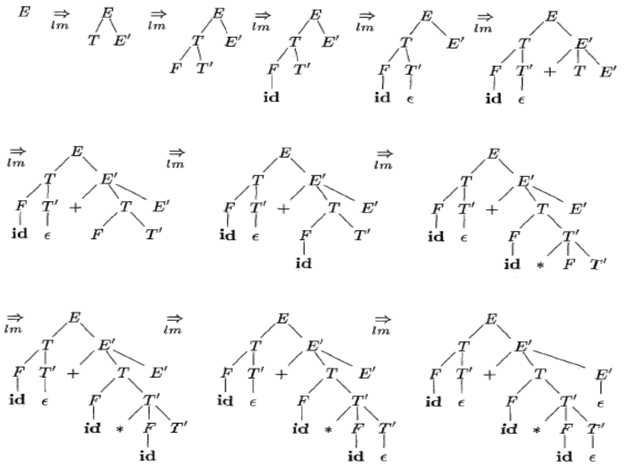
MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F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

Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$

Example

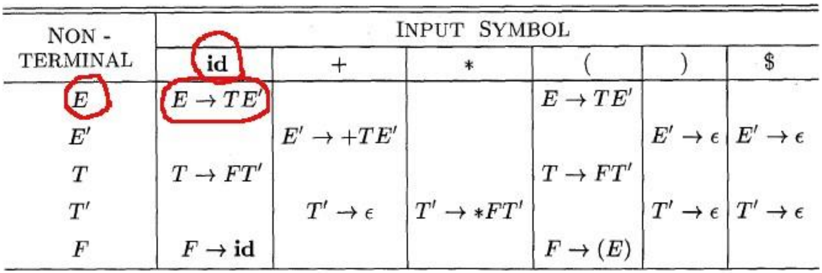
These moves correspond to a leftmost derivation,

$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} \mathbf{id}T'E' \underset{lm}{\Rightarrow} \mathbf{id}E' \underset{lm}{\Rightarrow} \mathbf{id} + TE' \underset{lm}{\Rightarrow} \cdots$$



Top-down parse for **id** + **id** ∗ **id**

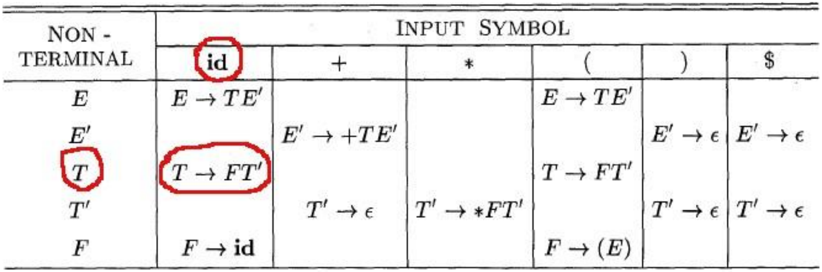
Example — *continued*



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

STACK	INPUT	ACTION
$E\$$	**id +id *id**\$	
↑	↑	
$TE'\ \$$	**id +id *id**\$	output $E \longrightarrow\ TE'$

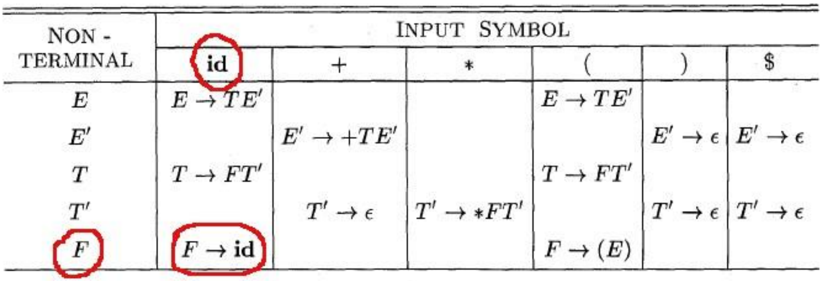
Example — *continued*



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

STACK	INPUT	ACTION
$TE'\$$	**id** +**id** ∗**id**$	
$FT'E'\$$	**id** +**id** ∗**id**$	output $T \to FT'$

Example — *continued*



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

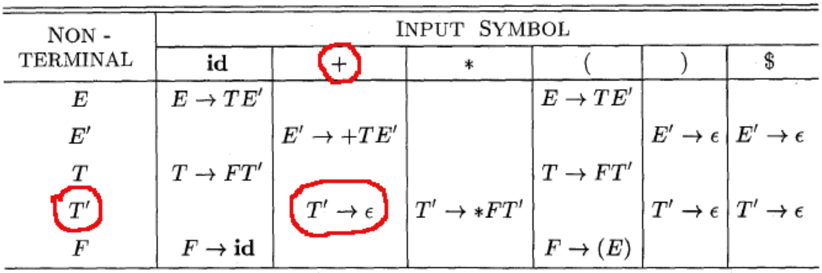
STACK	INPUT	ACTION
$FT'E'\$$	**id** +**id** ***id**$	
id$T'E'\$$	**id** +**id** ***id**$	output $F \to \textbf{id}$

Example — *continued*

STACK	INPUT	ACTION
id$T'E'$\$ ↑	**id** +**id** $*$**id**\$ ↑	match **id**
$T'E'$\$	+**id** $*$**id**\$ ↑	

Both are terminals and match. So, popped from the stack and input pointer advanced

Example — *continued*



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

STACK	INPUT	ACTION
$T'E'\$$	**+id *id**$	
$\uparrow$	$\uparrow$	
$E'\$$	**+id *id**$	output $T' \to \epsilon$

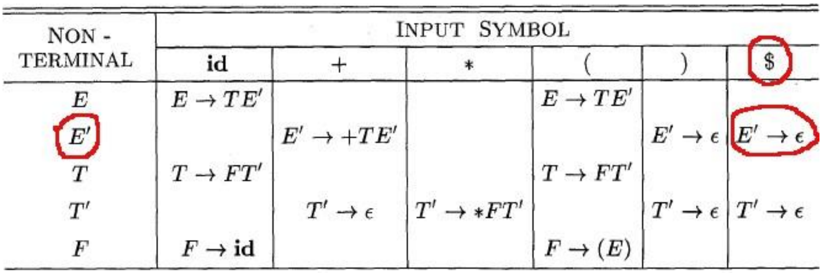
Example — *continued*

. . .

. . .

. . .

Example — *continued*



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

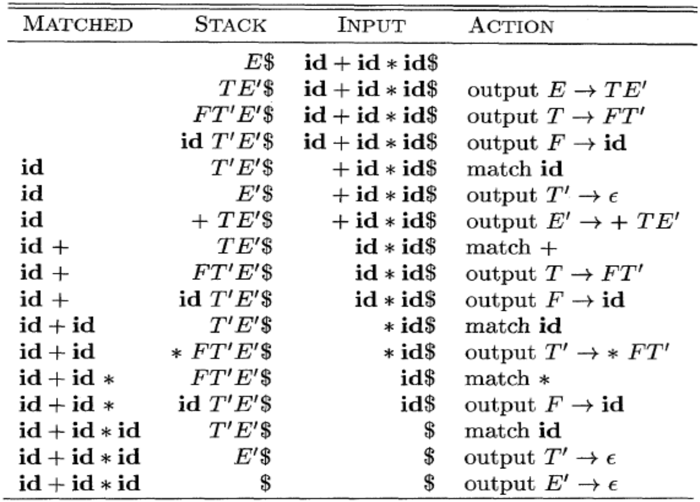
STACK	INPUT	ACTION
$E'\$$	$	
↑	↑	
$	$	output $E' \to \epsilon$

Example — *continued*

STACK	INPUT	ACTION
\$ ↑	\$ ↑	

Both are \$, the parser halts and announces successful completion of parsing.

Example



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 $E' \to \epsilon$

Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$

For a leftmost derivation the production rules in the ACTION column (outputs only) are to be used from top to bottom.

Thank you