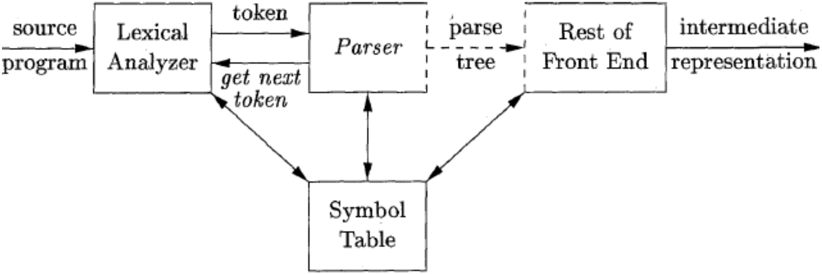
Syntax Analysis

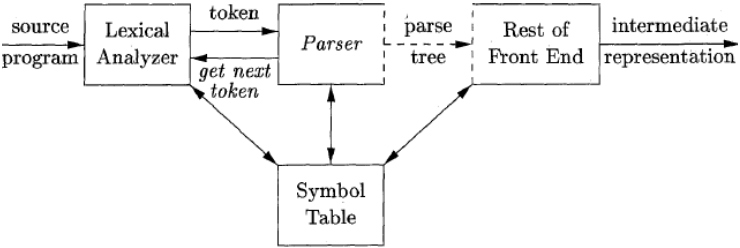
The Role of the Parser

- 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

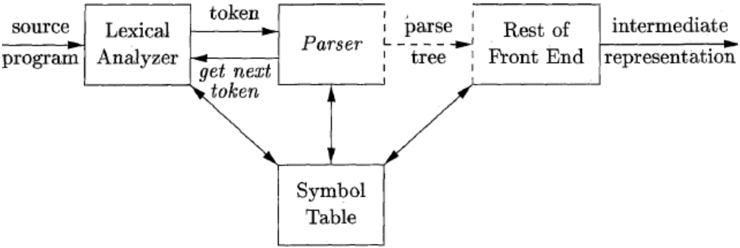
The Role of the Parser — *continued*



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.

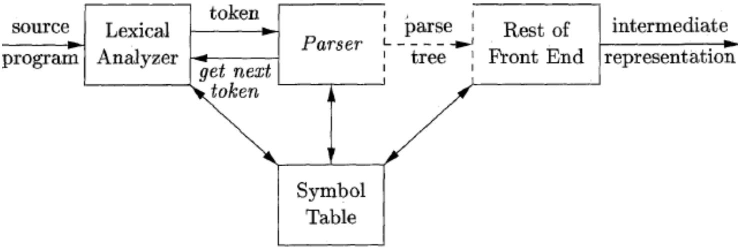
The Role of the Parser — *continued*



Position of parser in compiler model

- Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.
- In fact, the parse tree need not be constructed explicitly.
- Since checking and translation actions can be interspersed with parsing.

The Role of the Parser — *continued*



Position of parser in compiler model

- Thus, the parser and the rest of the front end could well be implemented by a single module.

The Role of the Parser — *continued*

- There are three general types of parsers for grammars:
  - universal,
  - top-down, and
  - bottom-up.
- Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar.
- 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.

The Role of the Parser — *continued*

- The most efficient top-down and bottom-up methods work only for subclasses of grammars.
- But several of these classes, particularly, LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages.
- Parsers implemented by hand often use LL grammars.
- The predictive-parsing approach works for LL grammars.
- Parsers for the larger class of LR grammars are usually constructed using automated tools.

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:

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

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

- 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:

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

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 to the use in a C program of the assignment operator $=$ instead of the comparison operator $==$.

The program containing $=$ may be well formed; however, it may not reflect the programmer's intent.

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*

- Another reason for emphasizing error recovery during parsing is that many errors appear syntactic, whatever their cause, and are exposed when parsing cannot continue.
- A few semantic errors, such as type mismatches, can also be detected efficiently.
- However, accurate detection of semantic and logical errors at compile time is in general a difficult task.

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.

Syntax Error Handling — *continued*

- Fortunately, common errors are simple ones.
- A relatively straightforward error-handling mechanism often suffices.
- How should an error handler report the presence of an error?
- At the very least, it must report the place in the source program where an error is detected.
- There is a good chance that the actual error occurred within the previous few tokens.
- A common strategy is to print the offending line with a pointer to the position at which an error is detected.

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 $A \rightarrow A\alpha \mid \beta$ can be replaced by the non-left-recursive productions

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

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

Example

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

to be replaced by

$$\begin{array}{rcl} A & \to & \beta A' \\ A' & \to & \alpha A' \mid \epsilon \end{array}$$

- Grammar for arithmetic expressions,

$$\begin{array}{rcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (\ E\ ) \mid \textbf{id} \end{array}$$

- Eliminating the immediate left recursions we obtain,

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

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,

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ \ldots \ \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \ \beta_n$$

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

$$\begin{array}{rcl} A & \rightarrow & \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ \ldots \ \mid \beta_n A' \\ A' & \rightarrow & \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ \ldots \ \mid \alpha_m A' \mid \epsilon \end{array}$$

- 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,

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

- 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, ..., 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 → A_j γ by
              the productions A_i → δ_1 γ | δ_2 γ | ... | δ_k γ
              where A_j → δ_1 | δ_2 | ... δ_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 $A \stackrel{+}{\Rightarrow} A$ 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 $A_i \to A_j\gamma$ 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) }

- 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.

arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
for (each i from 1 to n) {
  for (each j from 1 to $i - 1$) {
    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
  }
  eliminate the immediate left recursion among
    the A_i-productions;
}

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

```
1) arrange the nonterminals in some order A_1, A_2, ..., 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 → A_j γ by
              the productions A_i → δ_1 γ | δ_2 γ | ... | δ_k γ
              where A_j → δ_1 | δ_2 | ... δ_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 $A_i \to A_j\gamma$ 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) }

- As a result, on the ith iteration, the inner loop of lines (3) through (5) progressively raises the lower limit in any production $A_i \to A_m\alpha$, 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 $A_i \rightarrow A_j\gamma$ 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) }

- 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 ϵ-productions.

- We apply the procedure to grammar,

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

- 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*

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.

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 $A_1, A_2, \ldots, A_n$.

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) }

- $i = 1$, $A_1 = S$
- $j = 1$ 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

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

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

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) }

- $i = 2$, $A_2 = A$
- $j = 1$ 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 $A_i \to A_j\gamma$ 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) }

- $i = 2$, $A_2 = A$
- $j = 1$ 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

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

- $i = 2$, $A_2 = A$, $j = 1$, $A_1 = S$
- 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

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

- $i = 2$, $A_2 = A$, $j = 1$, $A_1 = S$
- 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 $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$ where $A_j \to \delta_1 \mid \delta_2 \mid \dots \delta_k$ are all the current A_j-productions

Left-Recursive Grammar

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

- $i = 2$, $A_2 = A$, $j = 1$, $A_1 = S$
- We need to
  - put productions of the form $S \to \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$
  - 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 $A_i \to A_j\gamma$ 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

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

- $i = 2$, $A_2 = A$, $j = 1$, $A_1 = S$
- 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 $A_i \to A_j\gamma$ 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

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

- $i = 2, A_2 = A, j = 1, A_1 = S$
- 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

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

- $i = 2$, $A_2 = A$, $j = 1$, $A_1 = S$
- 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

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

- $S \rightarrow Aa \mid b$ 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 $A_i \to A_j\gamma$ 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

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

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

$$A \to Aad \mid bd$$

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) $\Rightarrow$ eliminate the immediate left recursion among the A_i-productions;
7) }

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

- 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,

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

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

- 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,

$$\begin{array}{lcl} A & \rightarrow & bdA' \mid A' \\ A' & \rightarrow & cA' \mid adA' \mid \epsilon \end{array}$$

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 $A_i \to A_j\gamma$ 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) }

Conceptual Technique Summary (Again)

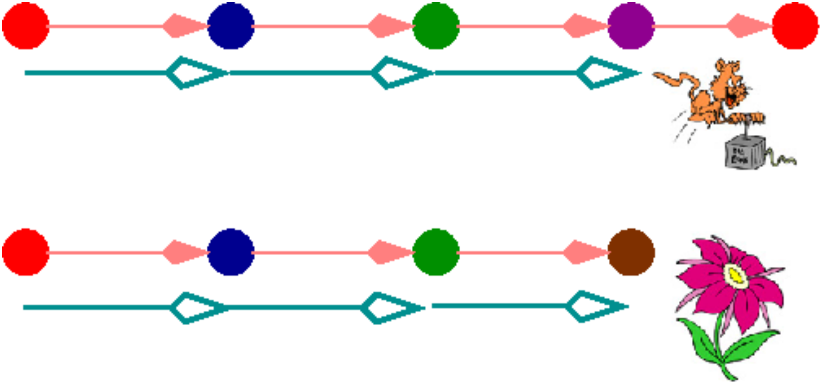
- Put some order in the nonterminals.
- Start by making first nonterminal productions left-recursion-free.
- Put the first nonterminal left-recursion-free productions into those of the second one.
- Now make the productions of second nonterminal left-recursion-free.
- Thus keep on growing the set of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

Left Factoring

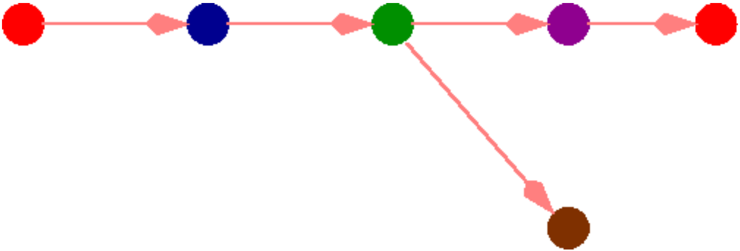
- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- The basic idea is that sometimes it is not clear which of two alternative productions to use to expand a nonterminal A.
- We may be able to rewrite the A-productions to defer the decision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

Left Factoring — *continued*

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



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



Left Factoring — *continued*

- We have the two productions,

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

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

Left Factoring — *continued*

- $A \to \alpha\beta_1 \mid \alpha\beta_2$ 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,

$$\begin{aligned} A &\to \alpha A' \\ A' &\to \beta_1 \mid \beta_2 \end{aligned}$$

Left Factoring Algorithm

INPUT. Grammar G.

OUTPUT An equivalent left-factored grammar.

Method.

- For each nonterminal A find the longest prefix α common to two or more of its alternatives.
- If $\alpha \neq \epsilon$ (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

$$\begin{array}{lcl} A & \rightarrow & \alpha A' \mid \gamma \\ A' & \rightarrow & \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{array}$$

  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*

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

- 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.
- Of course, both of the grammars are ambiguous.
- On input e, it will not be clear which alternative for S should be chosen.

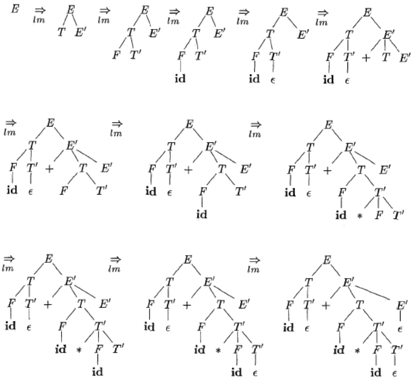
Top-Down Parsing

- 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

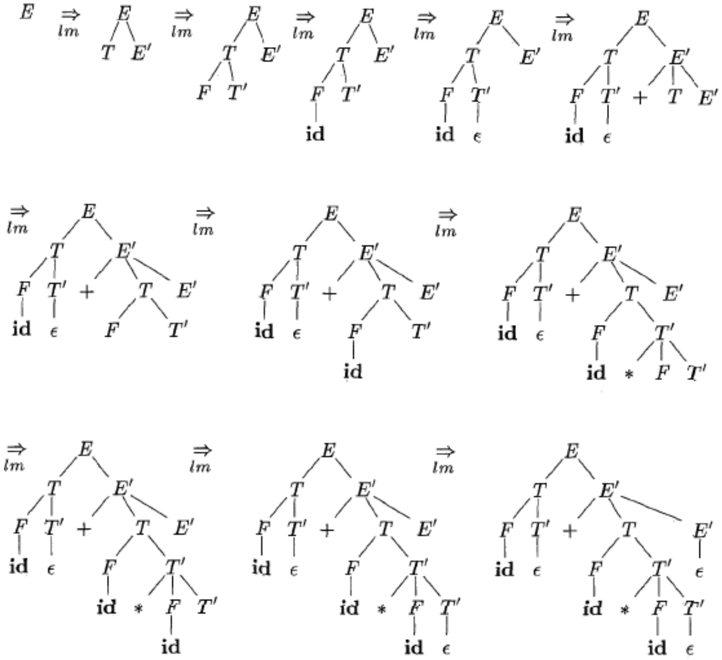
- The sequence of parse trees for the input **id** $+$ **id** $*$ **id** is a top-down parse according to grammar.

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



Top-down parse for $\mathbf{id} + \mathbf{id} * \mathbf{id}$

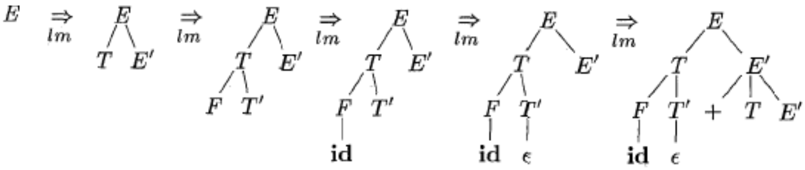
- This sequence of trees corresponds to a leftmost derivation of the input.



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.



- 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 $E' \rightarrow +TE'$ is chosen.
- At the second E' node, the production $E' \rightarrow \epsilon$ 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.

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

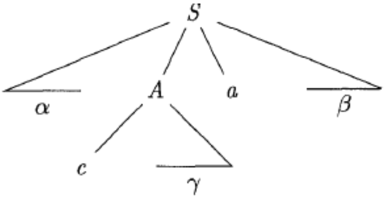
- The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G.
- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
- During panic-mode error recovery, sets of tokens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

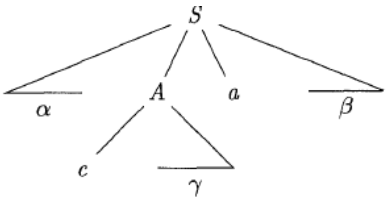


Terminal c is in FIRST(A) and a is in FOLLOW(A)

- Let us see how FIRST can be used during predictive parsing.
- Consider two *A*-productions $A \to \alpha \mid \beta$, where FIRST(α) and FIRST(β) 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 $A \to \beta$.

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 α, 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. If X is terminal, then FIRST(X) is $\{X\}$.

FIRST and FOLLOW — *continued*

2. - If X is nonterminal and $X \to Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), $\ldots$, FIRST(Y_{i-1}); that is, $Y_1, \ldots, Y_{i-1} \overset{*}{\Rightarrow} \epsilon$.
   - If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$ then add ϵ to FIRST(X).
3. If $X \to \epsilon$ is a production, then add ϵ to FIRST(X).

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

FIRST and FOLLOW — *continued*

Now, we can compute FIRST for any string $X_1 X_2 \ldots X_n$ as follows.

- Add to FIRST($X_1 X_2 \ldots X_n$) all the non-ϵ symbols of FIRST(X_1).
- Also add the non-ϵ symbols of FIRST(X_2) if ϵ is in FIRST(X_1), the non-ϵ symbols of FIRST(X_3) if ϵ is in both FIRST(X_1) and FIRST(X_2) and so on.
- Finally, add ϵ to FIRST($X_1 X_2 \ldots X_n$) if, for all i, ϵ is in FIRST(X_i).

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. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$ where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Example

1. If X is terminal, then FIRST(X) is $\{X\}$.
2. If $X \to \epsilon$ is a production, then add ϵ to FIRST(X).
3. If X is nonterminal and $X \to Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), $\ldots$, FIRST(Y_{i-1}); that is, $Y_1, \ldots, Y_{i-1} \overset{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$ then add ϵ to FIRST(X).

Grammar,

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

Then,

$$
\begin{aligned}
&\text{FIRST}(E) = \text{FIRST}(T) = \\
&\quad \text{FIRST}(F) = \{(, \mathbf{id}\} \\
&\text{FIRST}(E') = \{+, \epsilon\} \\
&\text{FIRST}(T') = \{*, \epsilon\}
\end{aligned}
$$

Example

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

- 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,

$$
\begin{aligned}
&\text{FIRST}(E) = \text{FIRST}(T) = \\
&\quad \text{FIRST}(F) = \{(, \textbf{id}\} \\
&\text{FIRST}(E') = \{+, \epsilon\} \\
&\text{FIRST}(T') = \{*, \epsilon\}
\end{aligned}
$$

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,

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

Then,

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

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

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

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

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

Grammar,

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

Then,

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

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

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

Example — *continued*

- Grammar:

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

- Computation of FOLLOW:

FOLLOW(E)	FOLLOW(E')	FOLLOW(T)	FOLLOW(T')	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*

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

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

By rule (2) *(If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except for ϵ is placed in FOLLOW(B))* *applied to,*

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

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

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

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

$F \to (E)$: 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.

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 ϵ (i.e., $\beta \overset{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

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

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 $\epsilon \in$ 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 ϵ (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)
\$,)		+		*

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

\$,)	\$,)	+		*

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

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

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

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

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 ϵ (i.e., $\beta \stackrel{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

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

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 $\epsilon \in$ 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 ϵ (i.e., $\beta \stackrel{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

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

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 $\epsilon \in$ 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 ϵ (i.e., $\beta \stackrel{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

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

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 $\epsilon \in$ 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 ϵ (i.e., $\beta \overset{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

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

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 $\epsilon \in$ 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) — continued *(If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ (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)
\$,)	\$,)	+, \$,)	+, \$,)	*

$T' \rightarrow *FT'$ (also $\epsilon \in$ FIRST(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).

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

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

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

$T' \to *FT'$ (also $\epsilon \in$ FIRST(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).

Example — *continued*

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

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

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

$T' \rightarrow *FT'$ (also $\epsilon \in$ FIRST(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).

Example — *continued*

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

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

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

$T' \to *FT'$ (also $\epsilon \in$ FIRST(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).

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.

LL(1) Grammars — *continued*

- The class of LL(1) grammars is rich enough to cover most programming constructs.
- Although care is needed in writing a suitable grammar for the source language.
- For example, no left-recursive or ambiguous grammar can be LL(1).

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.
2. At most one of α and β can derive the empty string.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A).
   Likewise, $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

LL(1) Grammars — *continued*

1. For no terminal *a* do both α and β derive strings beginning with *a*.
2. At most one of α and β can derive the empty string.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(*A*).

   Likewise, $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

- The first two conditions are equivalent to the statement that FIRST(α) and FIRST(β) are disjoint sets.
- The third condition is equivalent to stating that if ϵ is in FIRST(β), then FIRST(α) and FOLLOW(*A*) are disjoint sets, and likewise if ϵ is in FIRST(α).

A Case of LL(1) Grammar where We can Use FIRST and FOLLOW

$S \rightarrow AB \mid EF$, $A \rightarrow CD \mid EF$, $B \rightarrow EF$, $C \rightarrow c$, $D \rightarrow dE \mid \epsilon$,
$E \rightarrow e$, $F \rightarrow f$

A Case of LL(1) Grammar where We can Use FIRST and FOLLOW

$S \to AB \mid EF$, $A \to CD \mid EF$, $B \to EF$, $C \to c$, $D \to dE \mid \epsilon$,
$E \to e$, $F \to f$

$\text{FIRST}(S) = \{c, e\}$, $\text{FIRST}(A) = \{c, e\}$, $\text{FIRST}(B) = \{e\}$,
$\text{FIRST}(C) = \{c\}$, $\text{FIRST}(D) = \{\epsilon, d\}$, $\text{FIRST}(E) = \{e\}$,
$\text{FIRST}(F) = \{f\}$

A Case of LL(1) Grammar where We can Use FIRST and FOLLOW

$S \rightarrow AB \mid EF$, $A \rightarrow CD \mid EF$, $B \rightarrow EF$, $C \rightarrow c$, $D \rightarrow dE \mid \epsilon$, $E \rightarrow e$, $F \rightarrow f$

FIRST(S) $= \{c, e\}$, FIRST(A) $= \{c, e\}$, FIRST(B) $= \{e\}$, FIRST(C) $= \{c\}$, FIRST(D) $= \{\epsilon, d\}$, FIRST(E) $= \{e\}$, FIRST(F) $= \{f\}$

FOLLOW(S) $= \{\$\}$, FOLLOW($A$) $= \{e\}$, FOLLOW(B) $= \{\$\}$, FOLLOW($C$) $= \{d, e\}$, FOLLOW(D) $= \{e\}$, FOLLOW(E) $= \{f\}$, FOLLOW(F) $= \{e, \$\}$

A Case of LL(1) Grammar where We can Use FIRST and FOLLOW

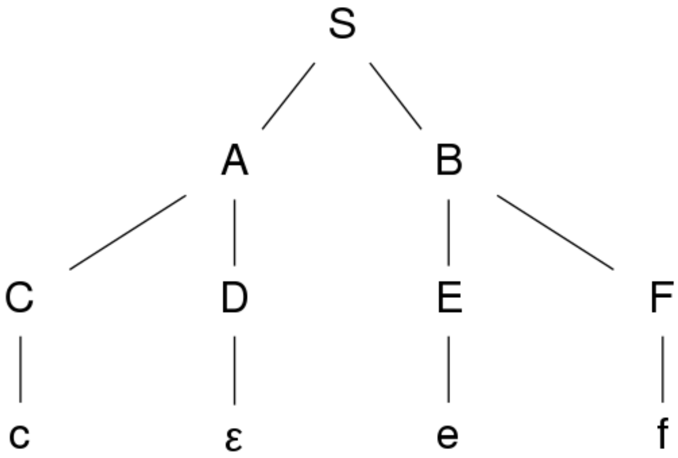
$S \rightarrow AB \mid EF$, $A \rightarrow CD \mid EF$, $B \rightarrow EF$, $C \rightarrow c$, $D \rightarrow dE \mid \epsilon$, $E \rightarrow e$, $F \rightarrow f$

FIRST$(S) = \{c, e\}$, FIRST$(A) = \{c, e\}$, FIRST$(B) = \{e\}$, FIRST$(C) = \{c\}$, FIRST$(D) = \{\epsilon, d\}$, FIRST$(E) = \{e\}$, FIRST$(F) = \{f\}$

FOLLOW$(S) = \{\$\}$, FOLLOW$(A) = \{e\}$, FOLLOW$(B) = \{\$\}$, FOLLOW$(C) = \{d, e\}$, FOLLOW$(D) = \{e\}$, FOLLOW$(E) = \{f\}$, FOLLOW$(F) = \{e, \$\}$

String to be parsed *cef*

A Case of LL(1) Grammar where We can Use FIRST and FOLLOW — *continued*



A Case of non LL(1) Grammar where We can not Use FIRST and FOLLOW

$S \to AB \mid EF$, $A \to CD \mid EF$, $B \to EF$, $C \to c$, $D \to eE \mid \epsilon$, $E \to e$, $F \to f$

A Case of non LL(1) Grammar where We can not Use FIRST and FOLLOW

$S \to AB \mid EF$, $A \to CD \mid EF$, $B \to EF$, $C \to c$, $D \to eE \mid \epsilon$, $E \to e$, $F \to f$

$\text{FIRST}(S) = \{c, e\}$, $\text{FIRST}(A) = \{c, e\}$, $\text{FIRST}(B) = \{e\}$,
$\text{FIRST}(C) = \{c\}$, $\text{FIRST}(D) = \{\epsilon, e\}$, $\text{FIRST}(E) = \{e\}$,
$\text{FIRST}(F) = \{f\}$

A Case of non LL(1) Grammar where We can not Use FIRST and FOLLOW

$S \to AB \mid EF$, $A \to CD \mid EF$, $B \to EF$, $C \to c$, $D \to eE \mid \epsilon$,
$E \to e$, $F \to f$

$\text{FIRST}(S) = \{c, e\}$, $\text{FIRST}(A) = \{c, e\}$, $\text{FIRST}(B) = \{e\}$,
$\text{FIRST}(C) = \{c\}$, $\text{FIRST}(D) = \{\epsilon, e\}$, $\text{FIRST}(E) = \{e\}$,
$\text{FIRST}(F) = \{f\}$

$\text{FOLLOW}(S) = \{\$\}$, $\text{FOLLOW}(A) = \{e\}$, $\text{FOLLOW}(B) = \{\$\}$,
$\text{FOLLOW}(C) = \{e\}$, $\text{FOLLOW}(D) = \{e\}$, $\text{FOLLOW}(E) = \{f\}$,
$\text{FOLLOW}(F) = \{e, \$\}$

A Case of non LL(1) Grammar where We can not Use FIRST and FOLLOW

$S \to AB \mid EF$, $A \to CD \mid EF$, $B \to EF$, $C \to c$, $D \to eE \mid \epsilon$, $E \to e$, $F \to f$

$\text{FIRST}(S) = \{c, e\}$, $\text{FIRST}(A) = \{c, e\}$, $\text{FIRST}(B) = \{e\}$, $\text{FIRST}(C) = \{c\}$, $\text{FIRST}(D) = \{\epsilon, e\}$, $\text{FIRST}(E) = \{e\}$, $\text{FIRST}(F) = \{f\}$

$\text{FOLLOW}(S) = \{\$\}$, $\text{FOLLOW}(A) = \{e\}$, $\text{FOLLOW}(B) = \{\$\}$, $\text{FOLLOW}(C) = \{e\}$, $\text{FOLLOW}(D) = \{e\}$, $\text{FOLLOW}(E) = \{f\}$, $\text{FOLLOW}(F) = \{e, \$\}$

String to be parsed *cef*

LL(1) Grammars — *continued*

- Predictive parsers can be constructed for LL(1) grammars since the proper production to apply for a nonterminal can be selected by looking only at the current input symbol.
- Flow-of-control constructs, with their distinguishing key-words, generally satisfy the LL(1) constraints.

LL(1) Grammars — *continued*

- For instance, if we have the productions,

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if}\ (\ \textit{expr}\ )\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
 & | & \textbf{while}\ (\ \textit{expr}\ )\ \textit{stmt} \\
 & | & \{\ \textit{stmt\_list}\ \}
\end{array}
$$

  then the keywords **if**, **while**, and the symbol { tell us which alternative is the one that could possibly succeed if we are to find a statement.

LL(1) Grammars — *continued*

- The next algorithm collects the information from FIRST and FOLLOW sets into a predictive parsing table $M[A, a]$, a two dimensional array, where A is a nonterminal, and a is a terminal or the symbol \$, the input endmarker.
- The idea behind the algorithm is the following.
- Suppose $A \rightarrow \alpha$ is a production with a in FIRST(α).
- Then, the parser will expand A by α when the current input symbol is a.
- The only complication occurs when $\alpha = \epsilon$ or $\alpha \stackrel{*}{\Rightarrow} \epsilon$.
- In this case, we should again expand A by α if the current input symbol is in FOLLOW(A), or if the \$ on the input has been reached and \$ is in FOLLOW(A).

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

If, after performing the above, 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,

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

the algorithm produces the parsing table in figure.

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

- Blanks are error entries.
- Nonblanks indicate a production with which to expand a nonterminal.

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

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

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 $\text{FIRST}(+TE') = \{+\}$.
- Since $\text{FOLLOW}(E') = \{), \$\}$, 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.

Example

- The following grammar, which abstracts the dangling-else problem, is repeated here:
- The grammar,

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

- The grammar is ambiguous.
- On input e, it will not be clear which alternative for S' should be chosen.

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		$E \rightarrow b$				

Example — *continued*

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

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

- The entry for $M[S', e]$ contains both $S' \to eS$ and $S' \to \epsilon$, since $\text{FOLLOW}(S') = \{e, \$\}$

Example — *continued*

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

- The grammar is ambiguous and the ambiguity is manifested by a choice in what production to use when an *e* (**else**) is seen.
- We can resolve the ambiguity if we choose $S' \to eS$.
- This choice corresponds to associating **else**'s with the closest previous **then**'s.

Example — *continued*

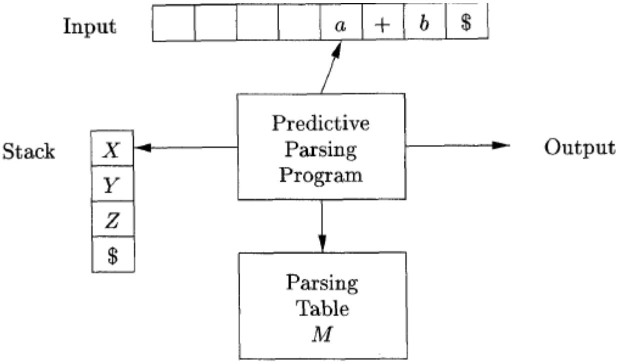
NON-TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

- Note that the choice $S' \to \epsilon$ would prevent *e* from ever being put on the stack or removed from the input, and is therefore surely wrong.

Nonrecursive Predictive Parsing

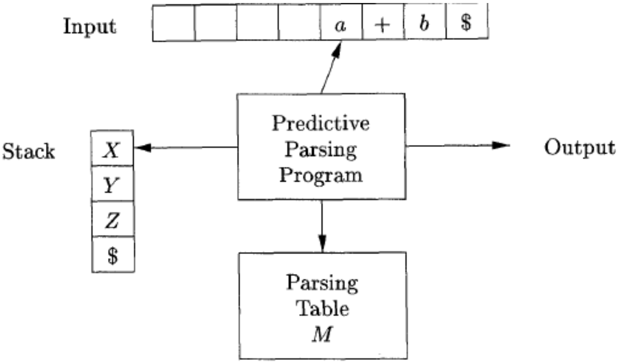
- A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls.
- The parser mimics a leftmost derivation.
- If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols a such that

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



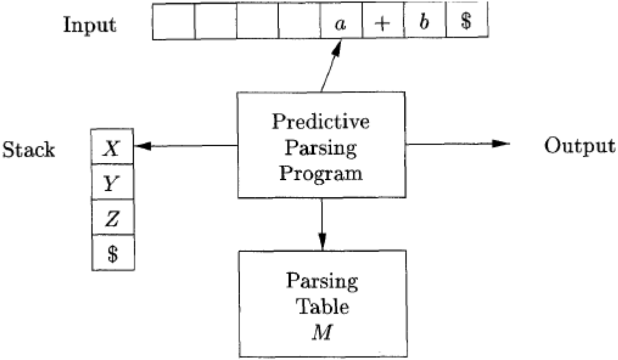
Model of a table-driven predictive parser

- 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.



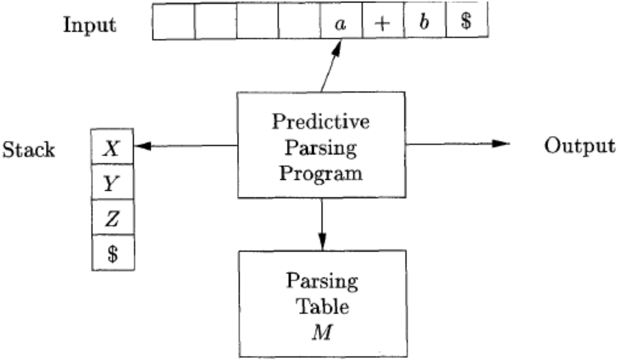
Model of a table-driven predictive parser

- 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 $.



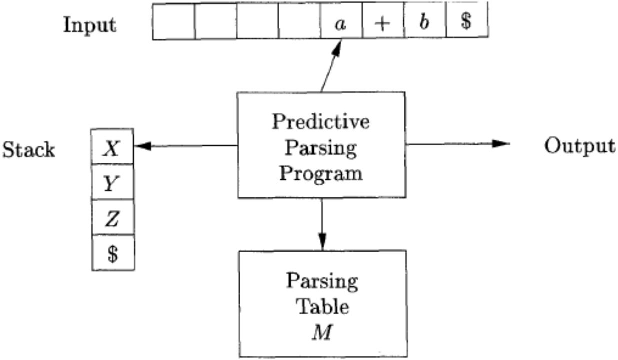
Model of a table-driven predictive parser

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



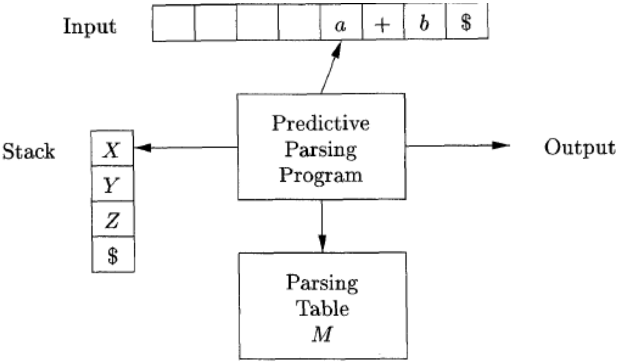
Model of a table-driven predictive parser

- 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.



Model of a table-driven predictive parser

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



Model of a table-driven predictive parser

- The behavior of the parser can be described in terms of its configurations, which give the stack contents and the remaining input.
- The next algorithm describes how configurations are manipulated.

Nonrecursive Predictive Parsing — *continued*

Nonrecursive predictive parsing.

INPUT. A string w and a parsing table M for grammar G.

OUTPUT.
- If w is in $L(G)$, a leftmost derivation of w;
- otherwise, an error indication.

METHOD.
- Initially, the parser is in a configuration in which it has $\$S$ on the stack with S, the start symbol of G on top, and w in the input buffer.
- The program that utilizes the predictive parsing table M to produce a parse for the input is shown here.

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
      if ( X is a ) pop the stack and advance ip;
      else if ( X is a terminal ) error();
      else if ( M[X, a] is an error entry ) error();
      else if ( M[X, a] = X → Y1 Y2 ··· Yk ) {
            output the production X → Y1 Y2 ··· Yk;
            pop the stack;
            push Yk, Yk-1, ... , Y1 onto the stack, with Y1 on top;
      }
      set X to the top stack symbol;
}
```

Predictive parsing algorithm

Example

- We consider grammar:

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

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

- 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 $\mathbf{id}+\mathbf{id}*\mathbf{id}$

- These moves correspond to a leftmost derivation,

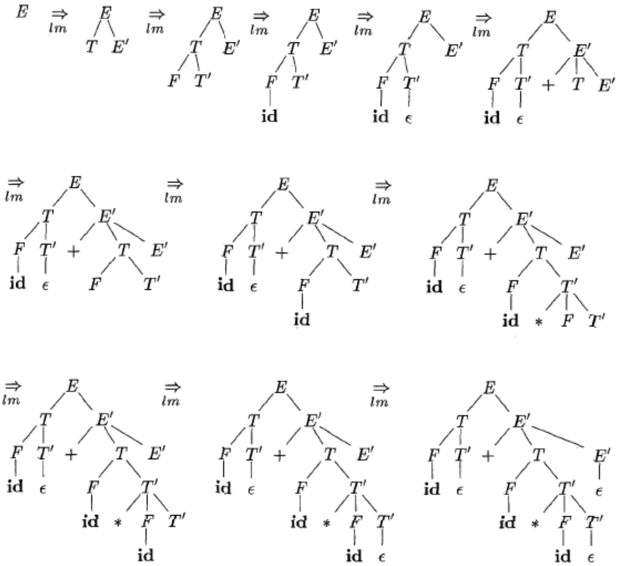
$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} \mathbf{id}\,T'E' \underset{lm}{\Rightarrow} \mathbf{id}\,E' \underset{lm}{\Rightarrow} \mathbf{id} + TE' \underset{lm}{\Rightarrow} \cdots$$

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

- 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 $\mathbf{id} + \mathbf{id} * \mathbf{id}$

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

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

- Note that the sentential forms in this derivation correspond to the input that has already been matched (in column MATCHED) followed by the stack contents.

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

- The matched input is shown only to highlight the correspondence.

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 the same reason, the top of the stack is to the left.
- When we consider bottom-up parsing, it will be more natural to show the top of the stack to the right.

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

- The input pointer points to the leftmost symbol of the string in the INPUT column.

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
$E\$$ $\uparrow$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$ $\uparrow$	
$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$

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
$TE'\$$ $\uparrow$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$ $\uparrow$	
$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	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	$F \to \mathbf{id}$			$F \to (E)$		

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

Example — *continued*

STACK	INPUT	ACTION
$\mathbf{id}T'E'\$$ $\uparrow$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$ $\uparrow$	match **id**

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

$T'E'\$$	$+\mathbf{id} * \mathbf{id}\$$ $\uparrow$	

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
$T'E'\$$ $\uparrow$	$+\mathbf{id} * \mathbf{id}\$$ $\uparrow$	
$E'\$$	$+\mathbf{id} * \mathbf{id}\$$	output $T \to \epsilon$

Example — *continued*

$\cdots$

$\cdots$

$\cdots$

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
$E'\$$ $\uparrow$	\$ $\uparrow$	
\$	\$	output $E' \to \epsilon$

Example — *continued*

STACK	INPUT	ACTION
\$ $\uparrow$	\$ $\uparrow$	

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

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.

Example — *continued*

$$
\begin{aligned}
E &\xRightarrow[E\to TE']{} TE' \\
&\xRightarrow[T\to FT']{} FT'E' \\
&\xRightarrow[F\to \mathbf{id}]{} \mathbf{id}\,T'E' \\
&\xRightarrow[T'\to \epsilon]{} \mathbf{id}\,E' \\
&\xRightarrow[E'\to +TE']{} \mathbf{id} + TE' \\
&\xRightarrow[T\to FT']{} \mathbf{id} + FT'E' \\
&\xRightarrow[F\to \mathbf{id}]{} \mathbf{id} + \mathbf{id}\,T'E'
\end{aligned}
$$

JT	ACTION
$*\,\mathbf{id}\$$	
$*\,\mathbf{id}\$$	output $E \to TE'$
$*\,\mathbf{id}\$$	output $T \to FT'$
$*\,\mathbf{id}\$$	output $F \to \mathbf{id}$
$*\,\mathbf{id}\$$	match **id**
$*\,\mathbf{id}\$$	output $T' \to \epsilon$
$*\,\mathbf{id}\$$	output $E' \to +\,TE'$
$*\,\mathbf{id}\$$	match $+$
$*\,\mathbf{id}\$$	output $T \to FT'$
$*\,\mathbf{id}\$$	output $F \to \mathbf{id}$
$*\,\mathbf{id}\$$	match **id**
$*\,\mathbf{id}\$$	output $T' \to *\,FT'$
$\mathbf{id}\$$	match $*$
$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\$$	match **id**
$\$$	output $T' \to \epsilon$
$\$$	output $E' \to \epsilon$

Example — *continued*

$$
\begin{aligned}
E &\xRightarrow[E\to TE']{} \;\cdots\cdots \\
&\xRightarrow[F\to \mathbf{id}]{} \mathbf{id} + \mathbf{id}\,T'E' \\
&\xRightarrow[T'\to *FT']{} \mathbf{id} + \mathbf{id} * FT'E' \\
&\xRightarrow[F\to \mathbf{id}]{} \mathbf{id} + \mathbf{id} * \mathbf{id}\,T'E' \\
&\xRightarrow[T'\to \epsilon]{} \mathbf{id} + \mathbf{id} * \mathbf{id}\,E' \\
&\xRightarrow[E'\to \epsilon]{} \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$

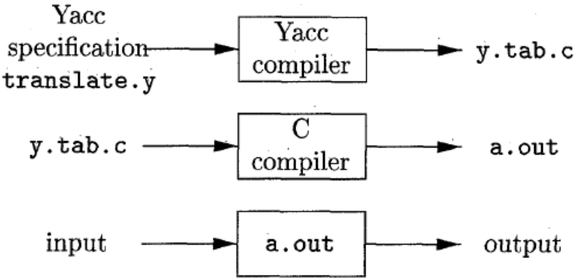
JT	ACTION
$*\,\mathbf{id}\$$	
$*\,\mathbf{id}\$$	output $E \to TE'$
$*\,\mathbf{id}\$$	output $T \to FT'$
$*\,\mathbf{id}\$$	output $F \to \mathbf{id}$
$*\,\mathbf{id}\$$	match **id**
$*\,\mathbf{id}\$$	output $T' \to \epsilon$
$*\,\mathbf{id}\$$	output $E' \to +\,TE'$
$*\,\mathbf{id}\$$	match $+$
$*\,\mathbf{id}\$$	output $T \to FT'$
$*\,\mathbf{id}\$$	output $F \to \mathbf{id}$
$*\,\mathbf{id}\$$	match **id**
$*\,\mathbf{id}\$$	output $T' \to *\,FT'$
$\mathbf{id}\$$	match $*$
$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\$$	match **id**
$\$$	output $T' \to \epsilon$
$\$$	output $E' \to \epsilon$

Parser Generators

- We show how a parser generator can be used to facilitate the construction of the front end of a compiler.
- `Yacc` stands for “yet another compiler-compiler,” reflecting the popularity of parser generators in the early 1970s when the first version of `Yacc` was created by S. C. Johnson.
- `Yacc` is available as a command on the UNIX system, and has been used to help implement many production compilers.

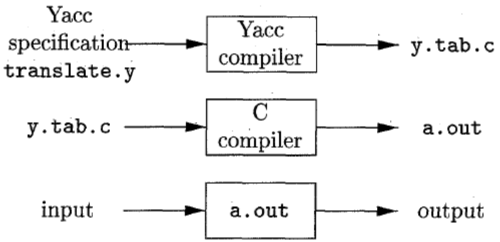
The Parser Generator `Yacc`

- A translator can be constructed using Yacc in the manner illustrated in figure.



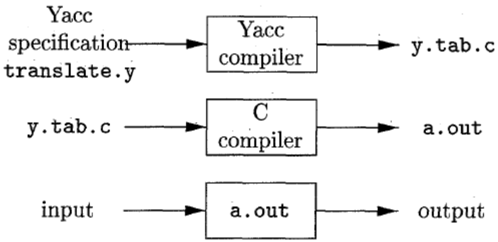
Creating an input/output translator with Yacc

The Parser Generator `Yacc` — *continued*



- First, a file, say `translate.y`, containing a `Yacc` specification of the translator is prepared.
- The UNIX system command `yacc translate.y` transforms the file `translate.y` into a C program called `y.tab.c`.
- The program `y.tab.c` is a representation of a parser written in C, along with other C routines that the user may have prepared.

The Parser Generator `Yacc` — *continued*



- By compiling `y.tab.c` along with the `ly` library that contains the LR parsing program using the command we obtain the desired object program `a.out` that performs the translation specified by the original `Yacc` program.
- If other procedures are needed, they can be compiled or loaded with `y.tab.c`, just as with any C program.

- A `Yacc` source program has three parts:

```
declarations
%%
translation rules
%%
supporting C routines
```

Example

- To illustrate how to prepare a `Yacc` source program, let us construct a simple desk calculator that reads an arithmetic expression, evaluates it, and then prints its numeric value.
- We shall build the desk calculator starting with the with the following grammar for arithmetic expressions:

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

- The token **digit** is a single digit between 0 and 9.

Example — *continued*

- A `Yacc` desk calculator program derived from this grammar is shown in figure.

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line    : expr '\n'         { printf("%d\n", $1); }
        ;
expr    : expr '+' term     { $$ = $1 + $3; }
        | term
        ;
term    : term '*' factor   { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'      { $$ = $2; }
        | DIGIT
        ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

Yacc specification of a simple desk calculator

The Parser Generator `Yacc` — *continued*

The Declarations Part

- There are two sections in the declarations part of a `Yacc` program.
- Both are optional.
- In the first section, we put ordinary C declarations, delimited by `%{` and `%}`.
- Here we piace declarations of any temporaries used by the translation rules or procedures of the second and third sections.

```
%{
#include <ctype.h>
%}

%token DIGIT
```

- This section contains only the include-statement

  `#include <ctype.h>`

  that causes the C preprocessor to include the standard header file `<ctype.h>` that contains the predicate `isdigit`.

```
%{
#include <ctype.h>
%}

%token DIGIT
```

- Also in the declarations part are declarations of grammar tokens.
- The statement

  `%token DIGIT`

  declares `DIGIT` to be a token.
- Tokens declared in this section can then be used in the second and third parts of the Yacc specification.
- If `Lex` is used to create the lexical analyzer that passes token to the `Yacc` parser, then these token declarations are also made available to the analyzer generated by `Lex`.

```
%{
#include <ctype.h>
%}

%token DIGIT
```

The Parser Generator Yacc — *continued*

The Translation Rules Part

```
%%
line    : expr '\n'          { printf("%d\n", $1); }
        ;
expr    : expr '+' term      { $$ = $1 + $3; }
        | term
        ;
term    : term '*' factor    { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'       { $$ = $2; }
        | DIGIT
        ;
```

- In the part of the `Yacc` specification after the first `%%` pair, we put the translation rules.
- Each rule consists of a grammar production and the associated semantic action.

- A set of productions that we have been writing:

  $$\langle\text{head}\rangle \rightarrow \langle\text{body}\rangle_1 \mid \langle\text{body}\rangle_2 \mid \cdots \mid \langle\text{body}\rangle_n$$

  would be written in Yacc as

$$\begin{array}{lcll} \langle\text{head}\rangle & : & \langle\text{body}\rangle_1 & \{\ \langle\text{semantic action}\rangle_1\ \} \\ & \mid & \langle\text{body}\rangle_2 & \{\ \langle\text{semantic action}\rangle_2\ \} \\ & & \ldots & \\ & \mid & \langle\text{body}\rangle_n & \{\ \langle\text{semantic action}\rangle_n\ \} \\ & ; & & \end{array}$$

The Parser Generator `Yacc` — *continued*

The Translation Rules Part

- In a `Yacc` production, unquoted strings of letters and digits not declared to be tokens are taken to be nonterminals.
- A quoted single character, e.g. `'c'`, is taken to be the terminal symbol `c`, as well as the integer code for the token represented by that character (i.e., Lex would return the character code for `'c'` to the parser, as an integer).
- Alternative bodies can be separated by a vertical bar, and a semicolon follows each head with its alternatives and their semantic actions.
- The first head is taken to be the start symbol.

```
%%
line    : expr '\n'           { printf("%d\n", $1); }
        ;
expr    : expr '+' term       { $$ = $1 + $3; }
        | term
        ;
term    : term '*' factor     { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'        { $$ = $2; }
        | DIGIT
        ;
```

- A `Yacc` semantic action is a sequence of C statements.
- In a semantic action, the symbol `$$` refers to the attribute value associated with the nonterminal of the head.

```
%%
line    : expr '\n'           { printf("%d\n", $1); }
        ;
expr    : expr '+' term       { $$ = $1 + $3; }
        | term
        ;
term    : term '*' factor     { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'        { $$ = $2; }
        | DIGIT
        ;
```

- While `$i` refers to the value associated with the *i*th grammar symbol (terminal or nonterminal) of the body.
- The semantic action is performed whenever we reduce by the associated production, so normally the semantic action computes a value for `$$` in terms of the `$i`'s.

- In the Yacc specification, we have written the two *E*-productions and their associated semantic actions.

$$
\begin{array}{lcl}
\boxed{E \rightarrow E + T \mid T} \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \textbf{digit}
\end{array}
$$

```
%%
line    : expr '\n'           { printf("%d\n", $1); }
        ;
expr    : expr '+' term       { $$ = $1 + $3; }
        | term
        ;
term    : term '*' factor     { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'        { $$ = $2; }
        | DIGIT
        ;
```

$$
\begin{array}{lcl}
\boxed{E \rightarrow E + T \mid T} & & \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \textbf{digit}
\end{array}
$$

```
%%
line    : expr '\n'           { printf("%d\n", $1); }
        ;
expr    : expr '+' term       { $$ = $1 + $3; }
        | term
        ;
term    : term '*' factor     { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'        { $$ = $2; }
        | DIGIT
        ;
```

- Note that the nonterminal `term` in the first production is the third grammar symbol of the body, while $+$ is the second.

$$
\begin{array}{lcl}
\boxed{E \rightarrow E + T \mid T} \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \textbf{digit}
\end{array}
$$

```
%%
line   : expr '\n'          { printf("%d\n", $1); }
       ;
expr   : expr '+' term      { $$ = $1 + $3; }
       | term
       ;
term   : term '*' factor    { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')'       { $$ = $2; }
       | DIGIT
       ;
```

- The semantic action associated with the first production adds the value of the `expr` and the `term` of the body and assigns the result as the value for the nonterminal `expr` of the head.

$$
\begin{array}{lcl}
\boxed{E \rightarrow E + T \mid T} \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \textbf{digit}
\end{array}
$$

```
%%
line    : expr '\n'          { printf("%d\n", $1); }
        ;
expr    : expr '+' term      { $$ = $1 + $3; }
        | term
        ;
term    : term '*' factor    { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'       { $$ = $2; }
        | DIGIT
        ;
```

- We have omitted the semantic action for the second production altogether, since copying the value is the default action for productions with a single grammar symbol in the body.
- In general, { `$$ = $1;` } is the default semantic action.

```
%%
line    : expr '\n'          { printf("%d\n", $1); }
        ;
expr    : expr '+' term      { $$ = $1 + $3; }
        | term
        ;
term    : term '*' factor    { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'       { $$ = $2; }
        | DIGIT
        ;
```

- Notice that we have added a new starting production `line : expr ’\n’ { printf ("\%d\n", $1); }` to the `Yacc` specification.
- This production says that an input to the desk calculator is to be an expression followed by a newline character.
- The semantic action associated with this production prints the decimal value of the expression followed by a newline character.

```
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

- The third part of a Yacc specification consists of supporting C-routines.

```
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

- A lexical analyzer by the name `yylex()` must be provided.
- Using `Lex` to produce `yylex()` is a common choice.
- Other procedures such as error recovery routines may be added as necessary.

```
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

- The lexical analyzer `yylex()` produces tokens consisting of a token name and its associated attribute value.
- If a token name such as `DIGIT` is returned, the token name must be declared in the first section of the `Yacc` specification.
- The attribute value associated with a token is communicated to the parser through a `Yacc`-defined variable `yylval`.

```
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

- The lexical analyzer here is very crude.
- It reads input characters one at a time using the C-function `getchar()`.
- If the character is a digit, the value of the digit is stored in the variable `yylval`, and the token name `DIGIT` is returned.
- Otherwise, the character itself is returned as the token name.

Creating `Yacc` Lexical Analyzers with `Lex`

- `Lex` was designed to produce lexical analyzers that could be used with `Yacc`.
- The Lex library `ll` will provide a driver program named `yylex()`, the name required by Yacc for its lexical analyzer.
- If `Lex` is used to produce the lexical analyzer, we replace the routine `yylex()` in the third part of the `Yacc` specification by the statement `#include "lex.yy.c"` and we have each `Lex` action return a terminal known to `Yacc`.
- By using the `#include "lex.yy.c"` statement, the program `yylex` has access to `Yacc`'s names for tokens, since the `Lex` output file is compiled as part of the Yacc output file `y.tab.c`.

Creating `Yacc` Lexical Analyzers with `Lex`

- `Lex` was designed to produce lexical analyzers that could be used with `Yacc`.
- The Lex library `ll` will provide a driver program named `yylex()`, the name required by Yacc for its lexical analyzer.
- If `Lex` is used to produce the lexical analyzer, we replace the routine `yylex()` in the third part of the `Yacc` specification by the statement `#include "lex.yy.c"` and we have each `Lex` action return a terminal known to `Yacc`.
- By using the `#include "lex.yy.c"` statement, the program `yylex` has access to `Yacc`'s names for tokens, since the `Lex` output file is compiled as part of the Yacc output file `y.tab.c`.

Creating `Yacc` Lexical Analyzers with `Lex` — *continued*

- Under the UNIX system, if the `Lex` specification is in the file `first.l` and the `Yacc` specification in `second.y`, we can say

  ```
  lex first.l
  yacc second.y
  cc y.tab.c -ly -ll
  ```

  to obtain the desired translator.

Creating Yacc Lexical Analyzers with Lex — *continued*

```
number   [0-9]+\e.?|[0-9]*\e.[0-9]+
%%
[ ]      { /* skip blanks */ }
{number} { sscanf(yytext, "%lf", &yylval);
                 return NUMBER; }
\n|.     { return yytext[0]; }
```

Lex specification for yylex()

- The Lex specification in figure can be used in place of the lexical analyzer for Yacc.
- The last pattern, meaning "any character," must be written `\n|.` since the dot in Lex matches any character except newline.

End of Slides