CS4212 - Compiler Design

Syntactic Analysis 2

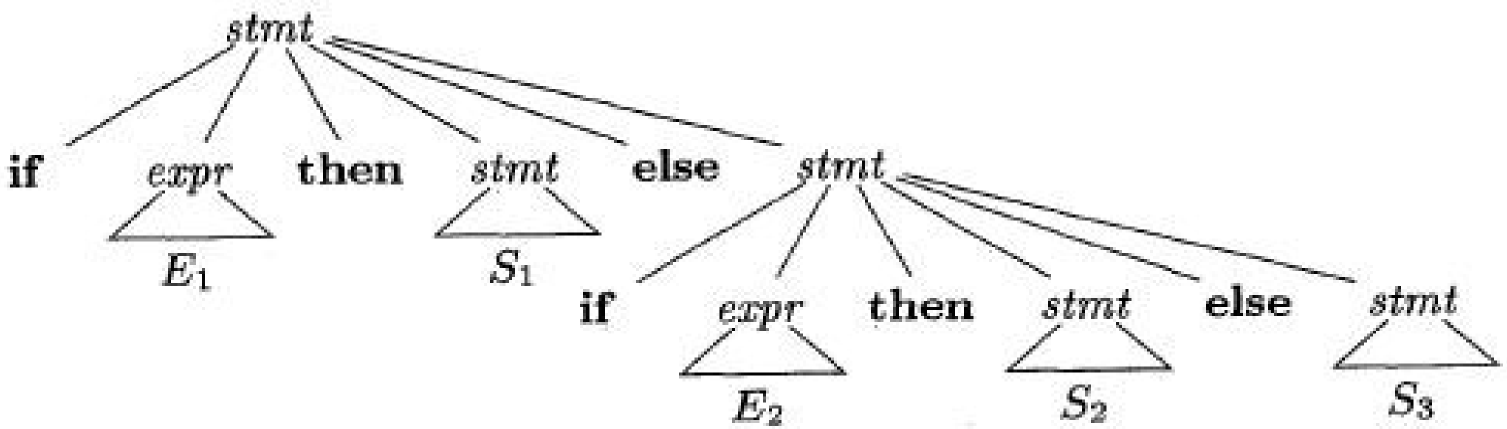
Outline

- Grammar transformations
- Top-down parsing
  - Recursive-descent parsing
  - Predictive parsing
  - LL(1) grammars
- Bottom-up parsing
  - Shift-reduce parsing
  - LR-parsing
- Yacc
- Textbook: "Compilers: Principles, Techniques, and Practices", Aho, Lam, Sethi, Ullman
  - Chapter 4: 4.3-4.7,4.9

Eliminating Ambiguity

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

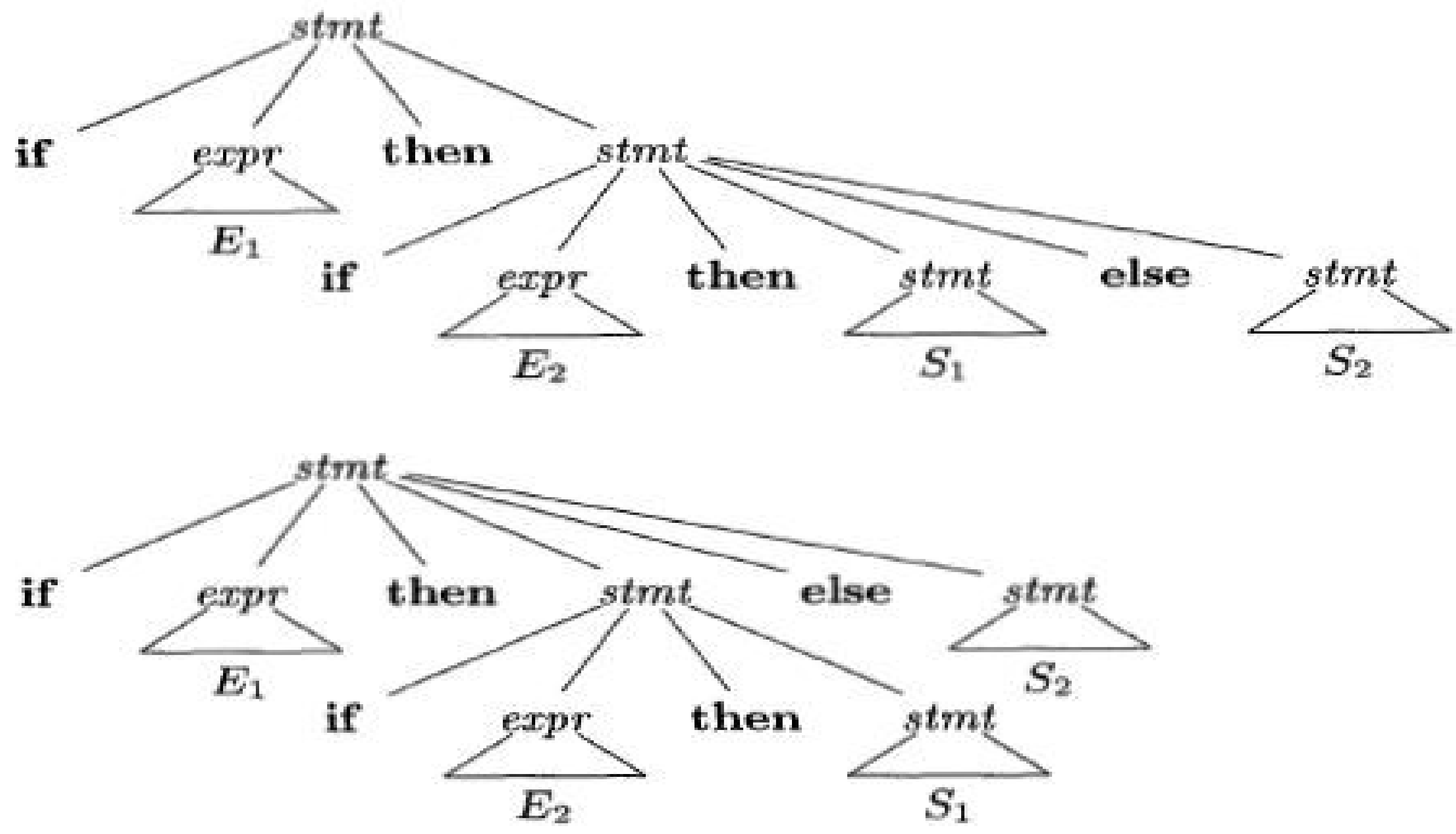
$$\textbf{if } E_1 \textbf{ then } S_1 \textbf{ else if } E_2 \textbf{ then } S_2 \textbf{ else } S_3$$



Eliminating Ambiguity

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

has the two parse trees



Elimination of Ambiguity

Unamibiguous grammar

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

Elimination of Left Recursion

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

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

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

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

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

Elimination of Left Recursion

Algorithm Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

```
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 current A_j-productions
5)          }
6)          eliminate the immediate left recursion among the A_i-productions
7)  }
```

Elimination of Left Recursion

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

Eliminating the immediate left recursion among these A-productions yields the following grammar.

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

Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

For example, if we have the two productions

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

on seeing the input **if**, we cannot immediately tell which production to choose to expand *stmt*. In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, and 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 to β_2. That is, left-factored, the original productions become

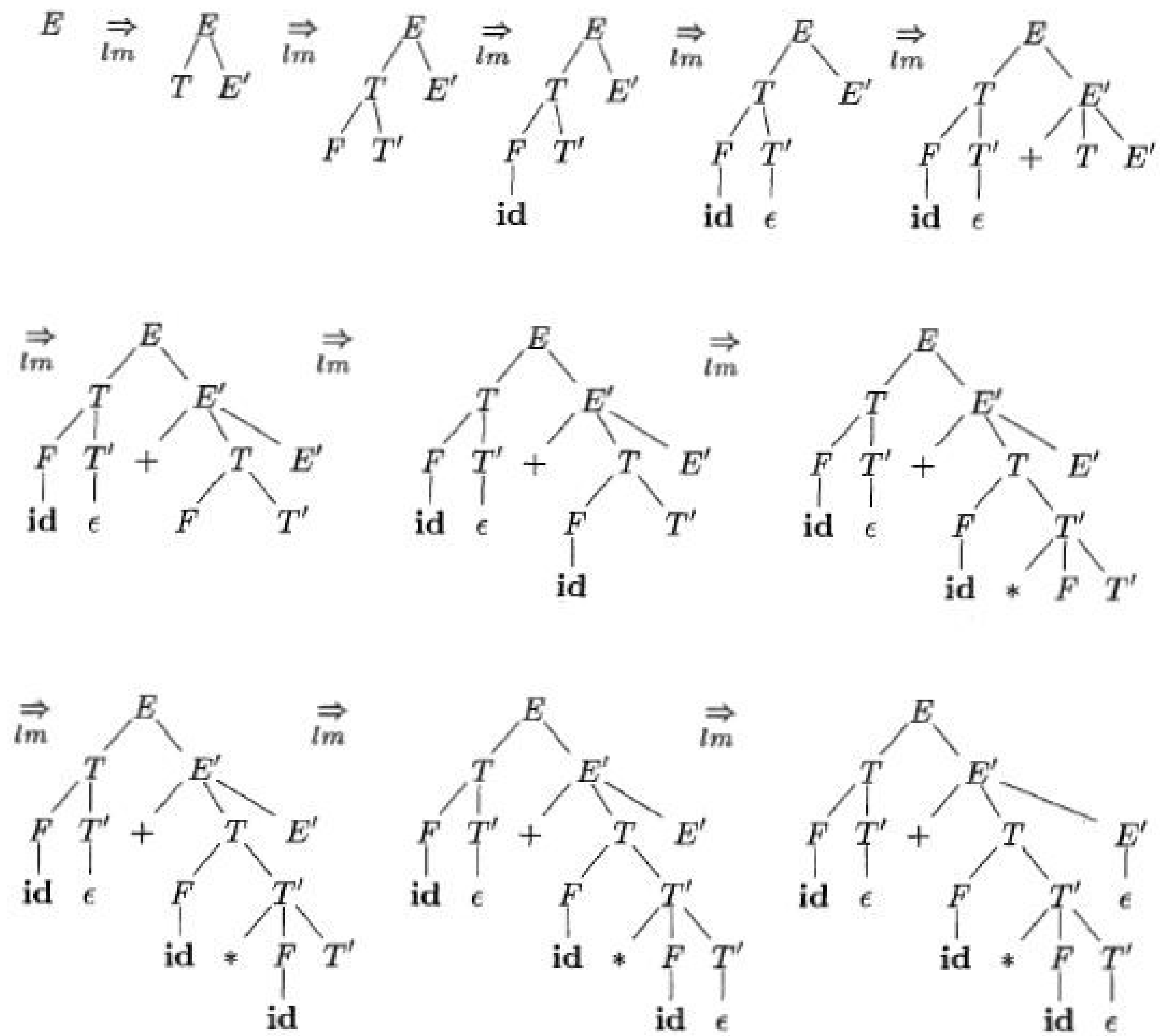
$$\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \end{array}$$

Top-Down Parsing

The Idea of Predictive Parsing

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

id+id*id



Recursive Descent Parsing

- One procedure for each non-terminal of the grammar.
- In the procedure, a production is chosen first.
  - Subsequent choices are performed by backtracking.
- For each symbol on rhs of production:
  - If symbol is terminal, match with input tape.
    - If match fails, backtrack.
    - If match succeeds, advance input tape
  - If symbol is non-terminal B, call B()
- Inefficient
  - Production is „guessed“
  - Wrong guess detected too late
  - Backtracking rewinds input tape, leading to multiple leads of same symbol
  - Worst case: exponential
- Does not work on left-recursive grammars

Recursive Descent Parsing

```
void A() {
        Choose an A-production, A → X_1 X_2 ··· X_k;
        for ( i = 1 to k ) {
                if ( X_i is a nonterminal )
                        call procedure X_i();
                else if ( X_i equals the current input symbol a )
                        advance the input to the next symbol;
                else /* an error has occurred */;
        }
}
```

FIRST and FOLLOW

Define $FIRST(\alpha)$, where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then ϵ is also in $\text{FIRST}(\alpha)$. For example, in Fig. 4.15, $A \stackrel{*}{\Rightarrow} c\gamma$, so c is in $\text{FIRST}(A)$.

Define $FOLLOW(A)$, for 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 $S \stackrel{*}{\Rightarrow} \alpha A a \beta$, for some α and β, as in Fig. 4.15. Note that there may have been symbols between A and a, at some time during the derivation, but if so, they derived ϵ and disappeared. In addition, if A can be the rightmost symbol in some sentential form, then \$ is in $\text{FOLLOW}(A)$; recall that \$ is a special "endmarker" symbol that is assumed not to be a symbol of any grammar.

FIRST

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

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

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

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

Now, we can compute FIRST for any string $X_1 X_2 \cdots X_n$ as follows. Add to FIRST($X_1 X_2 \cdots X_n$) all 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 FIRST(X_1) and FIRST(X_2); and so on. Finally, add ϵ to FIRST($X_1 X_2 \cdots X_n$) if, for all i, ϵ is in FIRST(X_i).

FOLLOW

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

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

1. $\text{FIRST}(F) = \text{FIRST}(T) = \text{FIRST}(E) = \{(, \mathbf{id}\}$. 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, $\text{FIRST}(T)$ must be the same as $\text{FIRST}(F)$. The same argument covers $\text{FIRST}(E)$.

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

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

Example

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

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

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

6. FOLLOW(F) = $\{+, *,), \$\}$. The reasoning is analogous to that for T in point (5).

LL(1) Grammars

A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ 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, if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, 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).

LL(1) Parsing Table

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

Blanks are error entries; nonblanks indicate a production with which to expand a non-terminal.

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

Predictive Parsing Machine

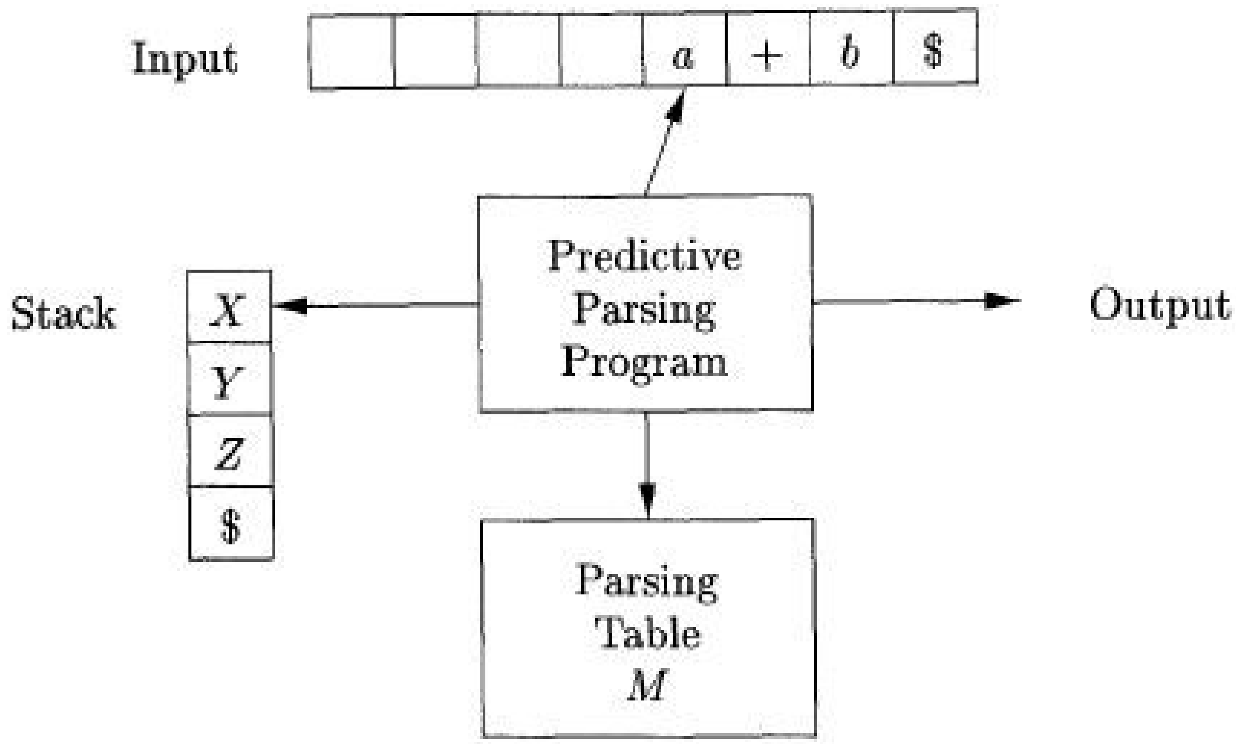


Figure 4.19: Model of a table-driven predictive parser

Predictive Parsing Algorithm

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 with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

```
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;
}
```

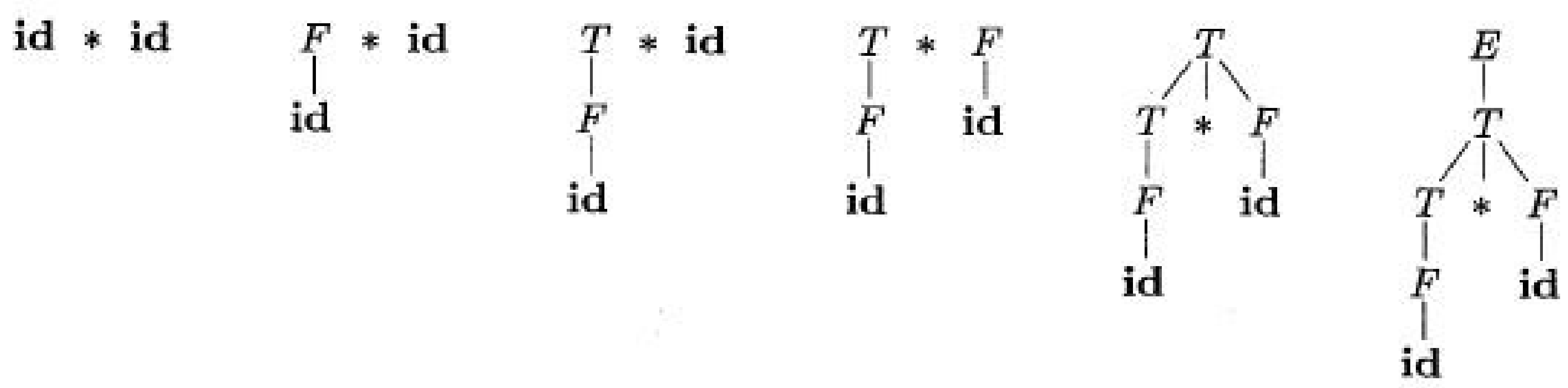
LL(1) Parsing - Discussion

- Caveats
  - can't work on left recursive grammars
    - left associativity is difficult to implement
  - difficult to add semantic actions
    - top-down parsing does not favor post-order traversal
  - more difficult to provide meaningful errors compared to bottom-up
- Tool: ANTLR (Java)
  - semantic actions encoded as virtual non-terminals – somewhat non-intuitive

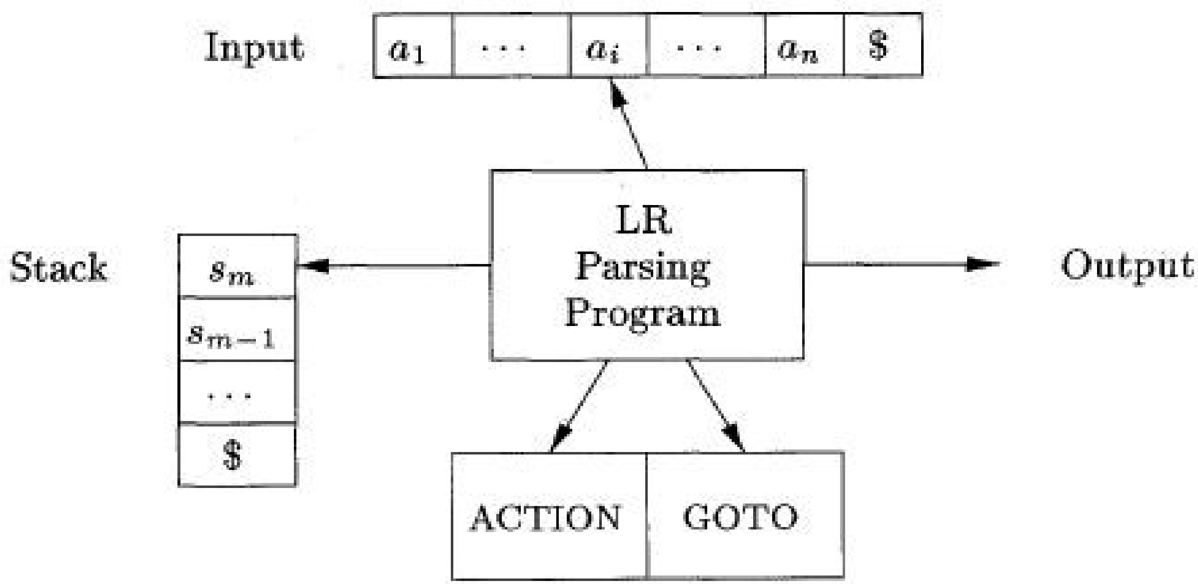
Bottom-up Parsing

Bottom-up = Leftmost

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$



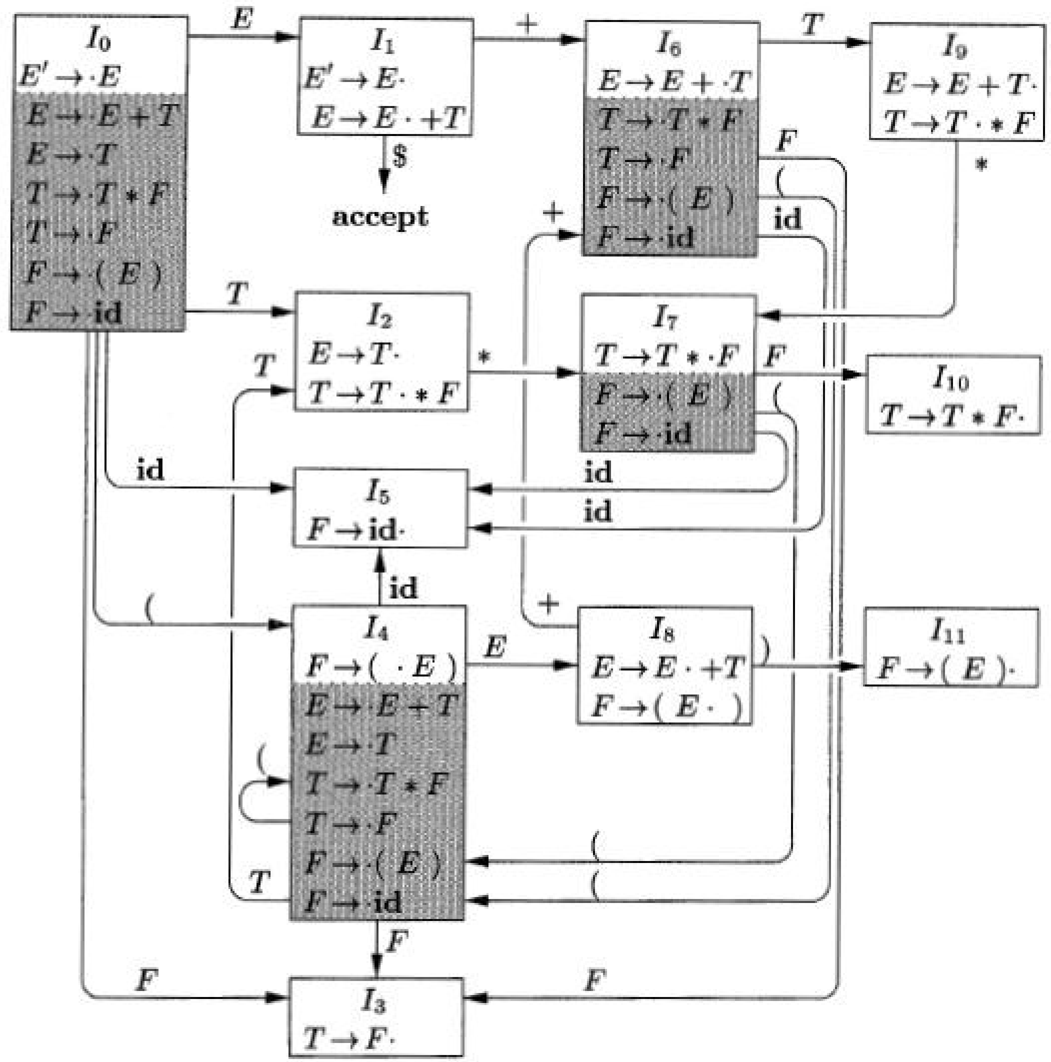
Shift-Reduce Parsing



Shift-Reduce Parsing

Stack	Input	Action
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \, \$$	shift
$\$ \, \mathbf{id}_1$	$* \, \mathbf{id}_2 \, \$$	reduce by $F \to \mathbf{id}$
$\$ \, F$	$* \, \mathbf{id}_2 \, \$$	reduce by $T \to F$
$\$ \, T$	$* \, \mathbf{id}_2 \, \$$	shift
$\$ \, T \, *$	$\mathbf{id}_2 \, \$$	shift
$\$ \, T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$ \, T * F$	$\$$	reduce by $T \to T * F$
$\$ \, T$	$\$$	reduce by $E \to T$
$\$ \, E$	$\$$	accept

LR(0) Parsing


I_0
$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_1
$E' \to E \cdot$
$E \to E \cdot + T$
$
accept
I_6
$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_9
$E \to E + T \cdot$
$T \to T \cdot * F$
I_2
$E \to T \cdot$
$T \to T \cdot * F$
I_7
$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_{10}
$T \to T * F \cdot$
I_5
$F \to \mathbf{id} \cdot$
I_4
$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_8
$E \to E \cdot + T$
$F \to (E \cdot)$
I_{11}
$F \to (E) \cdot$
I_3
$T \to F \cdot$
E
T
F
id
(
)
+
*

Yacc
(separate set of slides)