CS 4300: Compiler Theory

Chapter 4
Syntax Analysis

Xuejun Liang

2019 Fall

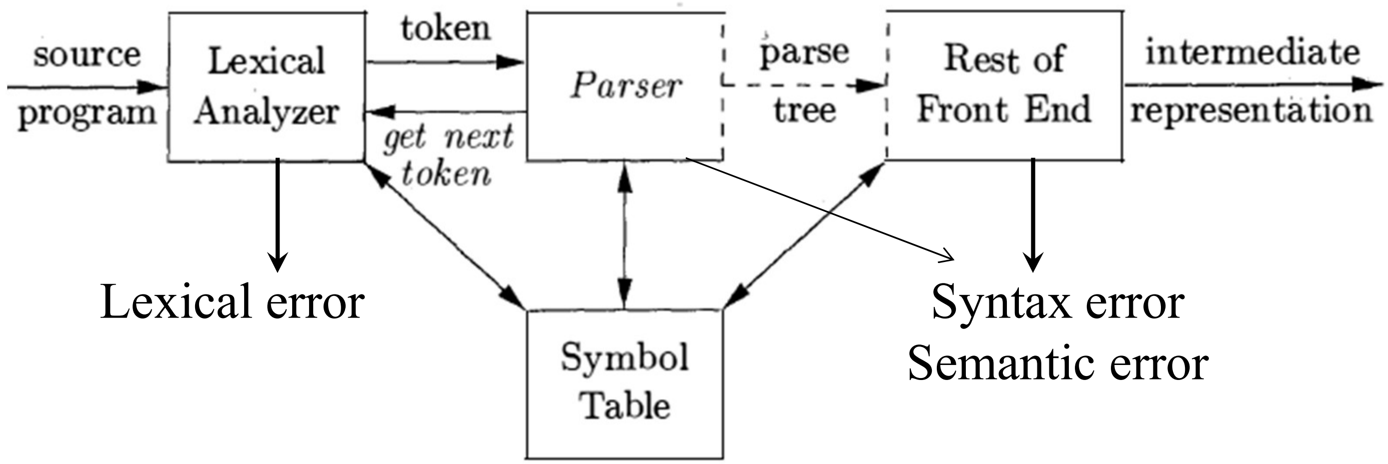
Outlines (Sections)

1. Introduction
2. Context-Free Grammars
3. Writing a Grammar
4. Top-Down Parsing
5. Bottom-Up Parsing
6. Introduction to LR Parsing: Simple LR
7. More Powerful LR Parsers
8. Using Ambiguous Grammars
9. Parser Generators

1. The role of the Parser

- A parser implements a Context-Free grammar as a recognizer of strings
- The role of the parser in a compiler is twofold:
  - To check syntax (= string recognizer)
    - And to report syntax errors accurately
  - To invoke semantic actions
    - For static semantics checking, e.g. type checking of expressions, functions, etc.
    - For syntax-directed translation of the source code to an intermediate representation

Position of Parser in Compiler Model



Error Handling

- A good compiler should assist in identifying and locating errors
  - *Lexical errors*: important, compiler can easily recover and continue
  - *Syntax errors*: most important for compiler, can almost always recover
  - *Static semantic errors*: important, can sometimes recover
  - *Dynamic semantic errors*: hard or impossible to detect at compile time, runtime checks are required
  - *Logical errors*: hard or impossible to detect

Viable-Prefix Property

- The *viable-prefix property* of parsers allows early detection of syntax errors
  - Goal: detection of an error *as soon as possible* without further consuming unnecessary input
  - How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language

Error is detected here ↓

Prefix {
```
...
for (;)
```
}
```
...
```

Error Recovery Strategies

- *Panic mode*
  - Discard input until a token in a set of designated synchronizing tokens (such as ;) is found.
- *Phrase-level recovery*
  - Perform local correction on the input to repair the error
- *Error productions*
  - Augment grammar with productions for erroneous constructs
- *Global correction*
  - Choose a minimal sequence of changes to obtain a global least-cost correction

Representative Grammars (Expression)

LR grammar

- Suitable for bottom-up parsing.
- Not suitable for top-down parsing
  - Because it is left recursive

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

LL grammar

- Non-left-recursive
- Suitable for top-down parsing

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

Ambiguous Grammar

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

2. Context-Free Grammars (Recap)

- Context-free grammar is a 4-tuple
  $G = (N, T, P, S)$ where
  - T is a finite set of tokens (*terminal* symbols)
  - N is a finite set of *nonterminals*
  - P is a finite set of *productions* of the form

    $$\alpha \rightarrow \beta$$

    where $\alpha \in (N \cup T)^* \, N \, (N \cup T)^*$ and $\beta \in (N \cup T)^*$
  - $S \in N$ is a designated *start symbol*

Notational Conventions

- Terminals
  $a,b,c,\ldots \in T$
  specific terminals: **0**, **1**, **id**, **+**
- Nonterminals
  $A,B,C,\ldots \in N$
  specific nonterminals: *expr*, *term*, *stmt*
- Grammar symbols
  $X,Y,Z \in (N \cup T)$
- Strings of terminals
  $u,v,w,x,y,z \in T^*$
- Strings of grammar symbols
  $\alpha,\beta,\gamma \in (N \cup T)^*$

Derivations (Recap)

- The *one-step derivation* is defined by
  $$\alpha\, A\, \beta \Rightarrow \alpha\, \gamma\, \beta$$
  where $A \rightarrow \gamma$ is a production in the grammar
- In addition, we define
  - $\Rightarrow$ is *leftmost* $\Rightarrow_{lm}$ if α does not contain a nonterminal
  - $\Rightarrow$ is *rightmost* $\Rightarrow_{rm}$ if β does not contain a nonterminal
  - Transitive closure $\Rightarrow^*$ (zero or more steps)
  - Positive closure $\Rightarrow^+$ (one or more steps)
- The *language generated by G* is defined by
  $$L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$$

Derivation (Example)

Grammar $G = (\{E\}, \{+,*,(,),-,\mathbf{id}\}, P, E)$ with productions P =

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

Example derivations:

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

$$E \Rightarrow_{rm} E + E \Rightarrow_{rm} E + \mathbf{id} \Rightarrow_{rm} \mathbf{id} + \mathbf{id}$$

$$E \Rightarrow^{*} E$$

$$E \Rightarrow^{*} \mathbf{id} + \mathbf{id}$$

$$E \Rightarrow^{+} \mathbf{id} * \mathbf{id} + \mathbf{id}$$

Language Classification

- A grammar G is said to be
  - *Regular* if it is *right linear* where each production is of the form
    $$A \to w\,B \qquad \text{or} \qquad A \to w$$
    or *left linear* where each production is of the form
    $$A \to B\,w \qquad \text{or} \qquad A \to w$$
  - *Context free* if each production is of the form
    $$A \to \alpha$$
    where $A \in N$ and $\alpha \in (N \cup T)^*$
  - *Context sensitive* if each production is of the form
    $$\alpha\, A\, \beta \to \alpha\, \gamma\, \beta$$
    where $A \in N$, $\alpha,\gamma,\beta \in (N \cup T)^*$, $|\gamma| > 0$
  - *Unrestricted*

Chomsky Hierarchy

$\mathtt{L}(\textit{regular}) \subset \mathtt{L}(\textit{context free}) \subset$
$\mathtt{L}(\textit{context sensitive}) \subset \mathtt{L}(\textit{unrestricted})$

Where $\mathtt{L}(T) = \{ L(G) \mid G \text{ is of type } T \}$
That is: the set of all languages
generated by grammars G of type T

Examples:

Every *finite language* is regular!
(construct a FSA for strings in $L(G)$)

$L_1 = \{ \mathbf{a}^n\mathbf{b}^n \mid n \geq 1 \}$ is context free, but not regular

$L_2 = \{ \mathbf{wcw} \mid \mathbf{w} \text{ is in } \mathrm{L}(\mathbf{a}|\mathbf{b})^* \}$ is context sensitive

$L_3 = \{ \mathbf{a}^n\mathbf{b}^m\mathbf{c}^n\mathbf{d}^m \mid n \geq 1 \}$ is context sensitive

3. Lexical Versus Syntactic Analysis

- Why use regular expressions to define the lexical syntax of a language?
  - Quite simple, more concise and easier-to-understand
  - More efficient lexical analyzers can be constructed automatically from regular expressions
  - Regular expressions are most useful for describing the structure of constructs such as identifiers, constants, keywords, and white space.
  - Grammars are most useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on.

Eliminating Ambiguity

if E1 **then if** E2 **then** S1 **else** S2

Ambiguous grammar: "dangling else"

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

Unambiguous grammar for if-then-else statements

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

Left Recursion

- A grammar is **left recursive** if it has a nonterminal A such that there is a derivation $A \stackrel{+}{\Rightarrow} A\ \alpha$ for some string α.
- When a grammar is left recursive then a predictive parser loops forever on certain inputs.
- **Immediate left recursion**, where there is a production of the form $A \to A\ \alpha$.

$$\begin{aligned} A &\to A\ \alpha \\ &\mid \beta \\ &\mid \gamma \end{aligned} \quad\Longrightarrow\quad \begin{aligned} A &\to \beta\ R \\ &\mid \gamma\ R \\ R &\to \alpha\ R \\ &\mid \varepsilon \end{aligned}$$

Algorithm to eliminate left recursion

Input: Grammar G with no cycles or ε-productions

Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$
for $i = 1, \ldots, n$ **{**
    for $j = 1, \ldots, i\text{-}1$ **{**
        replace each
            $A_i \to A_j\, \gamma$
        with
            $A_i \to \delta_1\, \gamma \mid \delta_2\, \gamma \mid \ldots \mid \delta_k\, \gamma$
        where
            $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
    }
    eliminate the *immediate left recursion* in A_i
}

Immediate Left-Recursion Elimination

Rewrite every left-recursive production

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

into a right-recursive production:

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

$$\begin{aligned} A &\to A\,\alpha \\ &\mid A\,\delta \\ &\mid \beta \\ &\mid \gamma \end{aligned} \quad \Rightarrow \quad \begin{aligned} A &\to \beta\, A' \\ &\mid \gamma\, A' \\ A' &\to \alpha\, A' \\ &\mid \delta\, A' \\ &\mid \varepsilon \end{aligned}$$

Example Left Recursion Elim.

$$\left.\begin{array}{l} S \to A\ a \mid b \\ A \to A\ c \mid S\ d \mid \epsilon \end{array}\right\}$$ Choose arrangement: *S*, *A*

i = 1: Nothing to do

i = 2, *j* = 1: *Replace S in A → S d with A a | b*

$$A \to A\ c \mid A\ a\ d \mid b\ d \mid \epsilon$$

Eliminate the *immediate left recursion* in *A*

$$\begin{array}{l} S \to A\ a \mid b \\ A \to b\ d\ A' \mid A' \\ A' \to c\ A' \mid a\ d\ A' \mid \epsilon \end{array}$$

Example Left Recursion Elim.

$$\left.\begin{array}{l} A \rightarrow B\ C \mid \mathbf{a} \\ B \rightarrow C\ A \mid A\ \mathbf{b} \\ C \rightarrow A\ B \mid C\ C \mid \mathbf{a} \end{array}\right\}\ \text{Choose arrangement: } A, B, C$$

$i = 1$: nothing to do

$i = 2, j = 1$: $B \rightarrow C\ A \mid \underline{A}\ \mathbf{b}$

$\Rightarrow \quad B \rightarrow C\ A \mid \underline{B\ C}\ \mathbf{b} \mid \underline{\mathbf{a}}\ \mathbf{b}$

$\Rightarrow_{\text{(imm)}} \quad B \rightarrow C\ A\ B_R \mid \mathbf{a}\ \mathbf{b}\ B_R$

$\quad B_R \rightarrow C\ \mathbf{b}\ B_R \mid \varepsilon$

$i = 3, j = 1$: $C \rightarrow \underline{A}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow \quad C \rightarrow \underline{B\ C}\ B \mid \underline{\mathbf{a}}\ B \mid C\ C \mid \mathbf{a}$

$i = 3, j = 2$: $C \rightarrow \underline{B}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow \quad C \rightarrow \underline{C\ A\ B_R}\ C\ B \mid \underline{\mathbf{a}\ \mathbf{b}\ B_R}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow_{\text{(imm)}} \quad C \rightarrow \mathbf{a}\ \mathbf{b}\ B_R\ C\ B\ C_R \mid \mathbf{a}\ B\ C_R \mid \mathbf{a}\ C_R$

$\quad C_R \rightarrow A\ B_R\ C\ B\ C_R \mid C\ C_R \mid \varepsilon$

Left Factoring

- When a nonterminal has two or more productions whose right-hand sides start with the same grammar symbols, the grammar is not LL(1) and cannot be used for predictive parsing
- Replace productions

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

  with

  $$A \rightarrow \alpha\ A_R \mid \gamma$$
  $$A_R \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

- Example:

$$S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a \quad \Rightarrow \quad \begin{array}{l} S \rightarrow i\ E\ t\ S\ S' \mid a \\ S' \rightarrow e\ S \mid \epsilon \end{array}$$

4. Top-Down Parsing

- Constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder
- Equivalently, finding the leftmost derivation for the input string

Grammar:

$E \rightarrow T + T$

$T \rightarrow$ **(** E **)**

$T \rightarrow$ **-** E

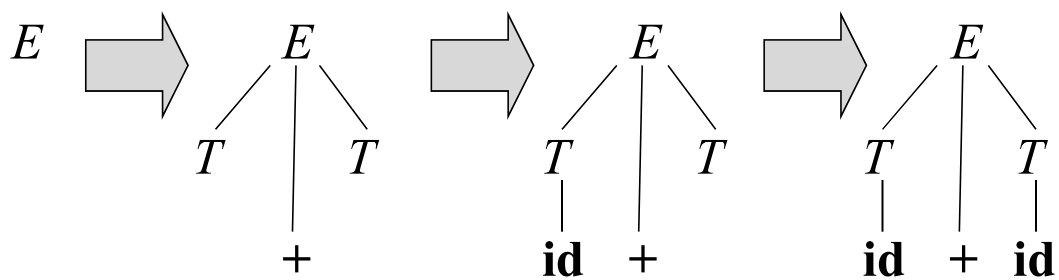
$T \rightarrow$ **id**

Leftmost derivation:

$E \Rightarrow_{lm} T + T$

$\Rightarrow_{lm}$ **id** $+ T$

$\Rightarrow_{lm}$ **id + id**



Parsing Methods

- *Universal* (any C-F grammar)
  - Cocke-Younger-Kasimi
  - Earley
- *Top-down* (C-F grammar with restrictions)
  - Recursive descent (predictive parsing)
  - LL (Left-to-right, Leftmost derivation) methods
- *Bottom-up* (C-F grammar with restrictions)
  - Operator precedence parsing
  - LR (Left-to-right, Rightmost derivation) methods
    - SLR, canonical LR, LALR

Predictive Parsing

- Eliminate left recursion from grammar
- Left factor the grammar
- Compute FIRST and FOLLOW
- Two variants:
  - Recursive (recursive-descent parsing)
  - Non-recursive (table-driven parsing)
- LL(k) class of grammars
  - It can be used to construct predictive parsers looking k symbols ahead in the input.

FIRST

- FIRST(α) = { *terminals that begin strings derived from* α }

  FIRST(a) = {a} if $a \in T$
  FIRST(ε) = {ε}
  FIRST(A) = $\cup_{A\to\alpha}$ FIRST(α) for $A\to\alpha \in P$
  FIRST($X_1X_2\ldots X_k$) =
  if for all $j = 1, \ldots, i$-1 : $\varepsilon \in$ FIRST(X_j) **then**
  add non-ε in FIRST(X_i) to FIRST($X_1X_2\ldots X_k$)
  if for all $j = 1, \ldots, k$: $\varepsilon \in$ FIRST(X_j) **then**
  add ε to FIRST($X_1X_2\ldots X_k$)

FOLLOW

- FOLLOW(A) = { *the set of terminals that can immediately follow nonterminal A* }

FOLLOW(A) =
  for all ($B \to \alpha\ A\ \beta$) $\in P$ **do**
    add FIRST(β)\{ε} to FOLLOW(A)
  for all ($B \to \alpha\ A\ \beta$) $\in P$ and $\varepsilon \in$ FIRST(β) **do**
    add FOLLOW(B) to FOLLOW(A)
  for all ($B \to \alpha\ A$) $\in P$ **do**
    add FOLLOW(B) to FOLLOW(A)
  if A is the start symbol S **then**
    add **$** to FOLLOW($A$)

Example

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

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

FIRST(E') = {+, ε}

FIRST(T') = {*, ε}

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

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

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

LL(1) Grammar

- Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1)
- A grammar G is LL(1) if it is not left recursive and for each collection of productions

  $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

  for nonterminal A the following holds:

  1. $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \varnothing$ for all $i \neq j$
  2. if $\alpha_i \Rightarrow^* \varepsilon$ then

     2.a. $\alpha_j \not\Rightarrow^* \varepsilon$ for all $j \neq i$

     2.b. $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \varnothing$ for all $j \neq i$

Non-LL(1) Examples

Grammar	*Not LL(1) because:*
$S \rightarrow S\,\mathbf{a} \mid \mathbf{a}$	Left recursive
$S \rightarrow \mathbf{a}\,S \mid \mathbf{a}$	FIRST($\mathbf{a}\,S$) $\cap$ FIRST($\mathbf{a}$) $\neq \varnothing$
$S \rightarrow \mathbf{a}\,R \mid \varepsilon$ $R \rightarrow S \mid \varepsilon$	For R: $S \Rightarrow^* \varepsilon$ and $\varepsilon \Rightarrow^* \varepsilon$
$S \rightarrow \mathbf{a}\,R\,\mathbf{a}$ $R \rightarrow S \mid \varepsilon$	For R: FIRST(S) $\cap$ FOLLOW(R) $\neq \varnothing$
$S \rightarrow \mathbf{i}\,E\,\mathbf{t}\,S\,S' \mid \mathbf{a}$ $S' \rightarrow \mathbf{e}\,S \mid \varepsilon$ $E \rightarrow \mathbf{b}$	For S': FIRST($\mathbf{e}\,S$) $\cap$ FOLLOW(S') $\neq \varnothing$

Using FIRST and FOLLOW in a Recursive-Descent Parser

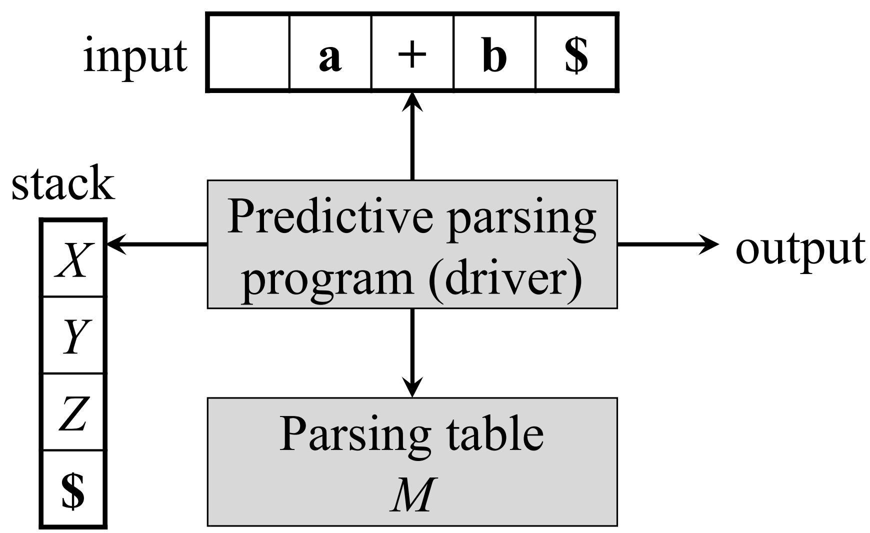
expr → *term rest*
rest → + *term rest*
| **-** *term rest*
| ε
term → **id**

procedure *rest*();
begin
  if *lookahead* in FIRST(+ *term rest*) **then**
    match(‘+’); *term*(); *rest*()
  else if *lookahead* in FIRST(**-** *term rest*) **then**
    match(‘**-**’); *term*(); *rest*()
  else if *lookahead* in FOLLOW(*rest*) **then**
    return
  else error**()**
end;

where FIRST(+ *term rest*) = { + }
FIRST(**-** *term rest*) = { **-** }
FOLLOW(*rest*) = { **$** }

Non-Recursive Predictive Parsing: Table-Driven Parsing

- Given an LL(1) grammar $G = (N, T, P, S)$ construct a table $M[A,a]$ for $A \in N$, $a \in T$ and use a *driver program* with a *stack*



Constructing an LL(1) Predictive Parsing Table

```
for each production A → α {
    for each a ∈ FIRST(α) {
        add A → α to M[A,a]
    }
    if ε ∈ FIRST(α) {
        for each b ∈ FOLLOW(A) {
            add A → α to M[A,b]
        }
    }
}
Mark each undefined entry in M error
```

Example Table

$$E \rightarrow T\,E'$$
$$E' \rightarrow +\,T\,E' \mid \varepsilon$$
$$T \rightarrow F\,T'$$
$$T' \rightarrow *\,F\,T' \mid \varepsilon$$
$$F \rightarrow (\,E\,) \mid \textbf{id}$$

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$E \rightarrow T\,E'$	**(id**	**$)**
$E' \rightarrow +\,T\,E'$	+	**$)**
$E' \rightarrow \varepsilon$	ε	
$T \rightarrow F\,T'$	**(id**	**+ $)**
$T' \rightarrow *\,F\,T'$	*	**+ $)**
$T' \rightarrow \varepsilon$	ε	
$F \rightarrow (\,E\,)$	**(**	*** + $)**
$F \rightarrow \textbf{id}$	**id**	*** + $)**

	id	+	*	**(**	**)**	**$**
E	$E \rightarrow T\,E'$			$E \rightarrow T\,E'$		
E'		$E' \rightarrow +\,T\,E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\,T'$			$T \rightarrow F\,T'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *\,F\,T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \textbf{id}$			$F \rightarrow (\,E\,)$		

LL(1) Grammars are Unambiguous

Ambiguous grammar
$S \rightarrow$ **i** E **t** $S\,S'$ | **a**
$S' \rightarrow$ **e** S | ε
$E \rightarrow$ **b**

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$S \rightarrow$ **i** E **t** $S\,S'$	**i**	**e $**
$S \rightarrow$ **a**	**a**	
$S' \rightarrow$ **e** S	**e**	**e $**
$S' \rightarrow \varepsilon$	ε	
$E \rightarrow$ **b**	**b**	**t**

Error: duplicate table entry

	a	**b**	**e**	**i**	**t**	**$**
S	$S \rightarrow$ **a**			$S \rightarrow$ **i** E **t** $S\,S'$		
S'			$S' \rightarrow \varepsilon$ $S' \rightarrow$ **e** S			$S' \rightarrow \varepsilon$
E		$E \rightarrow$ **b**				

Predictive Parsing Program (Driver)

```
read w$ into the input buffer; // w is the input
push($); push(S);
a = lookahead;            // set ip to point to the first symbol of w
X = pop();
while ( X ≠ $ ) {
    if ( X is a ) a = lookahead;          // advance ip;
    else if ( X is a terminal ) error();
    else if ( M [X, a] is an error entry ) error();
    else if ( M[X, a] = X → Y1Y2 …Yk ) {
        output the production X → Y1Y2 …Yk ;
        push (Yk); push(Yk–1) , ... , push(Y1);
    }
    X = pop();
}
```

Example: Moves of table-driven parsing on input **id + id * id**

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

Panic Mode Recovery

Add synchronizing actions to undefined entries based on FOLLOW

Pro: Can be automated
Cons: Error messages are needed

FOLLOW(E) = { **) $** }
FOLLOW(E') = { **) $** }
FOLLOW(T) = { **+) $** }
FOLLOW(T') = { **+) $** }
FOLLOW(F) = { **+ *) $** }

	id	**+**	*****	**(**	**)**	**$**
E	$E \to T E'$			$E \to T E'$	***synch***	***synch***
E'		$E' \to + T E'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to F T'$	***synch***		$T \to F T'$	***synch***	***synch***
T'		$T' \to \varepsilon$	$T' \to * F T'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ **id**	***synch***	***synch***	$F \to$ **(** E **)**	***synch***	***synch***

synch: the driver pops current nonterminal A and skips input till synch token or skips input until one of FIRST(A) is found

Example: Moves of parsing and error recovery on the erroneous input ***) id * + id***

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

Phrase-Level Recovery

Change input stream by inserting missing tokens
For example: **id id** is changed into **id * id**

Pro:	Can be fully automated
Cons:	Recovery not always intuitive

Can then continue here

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow T E'$			$E \rightarrow T E'$	***synch***	***synch***
E'		$E' \rightarrow + T E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F T'$	***synch***		$T \rightarrow F T'$	***synch***	***synch***
T'	***insert ****	$T' \rightarrow \varepsilon$	$T' \rightarrow * F T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ **id**	***synch***	***synch***	$F \rightarrow (E)$	***synch***	***synch***

insert *: driver inserts missing ***** and retries the production