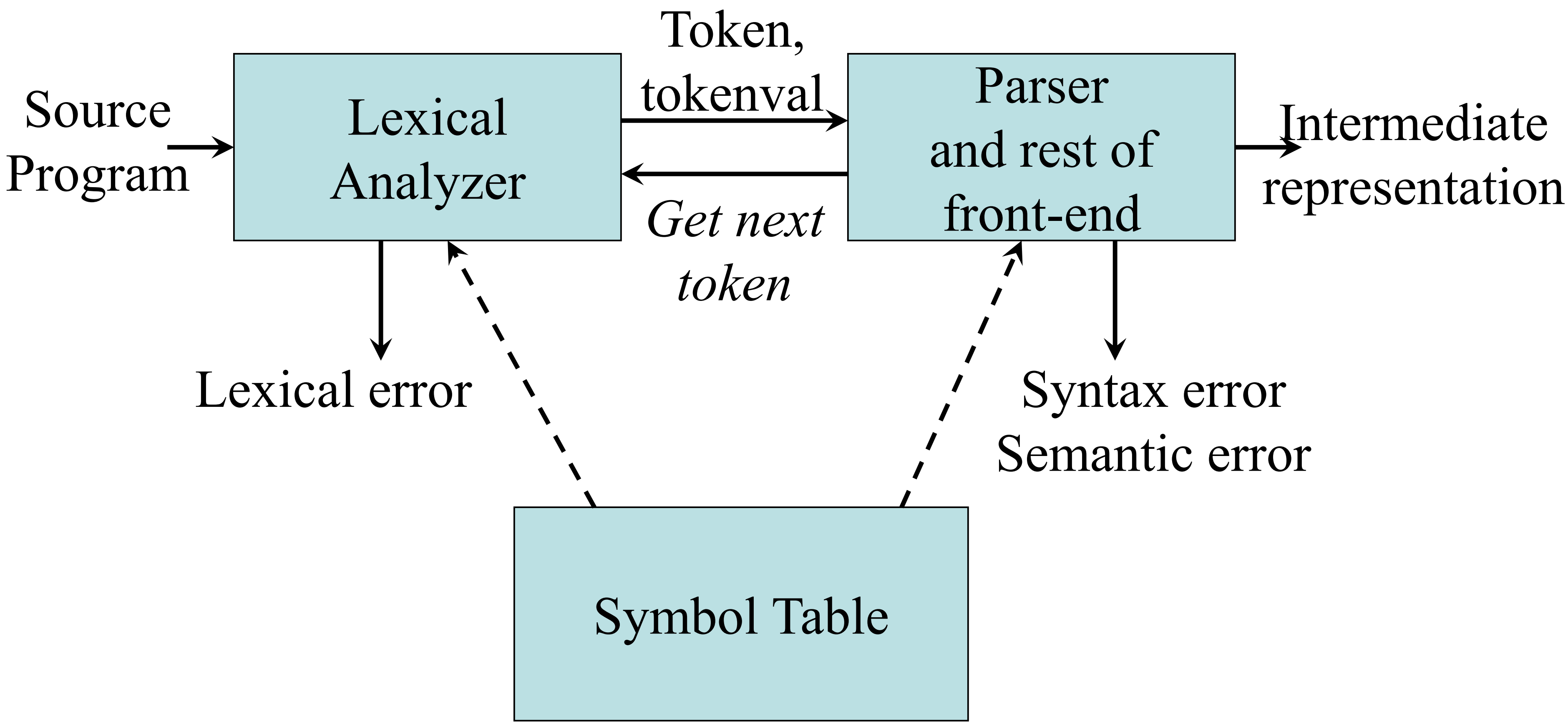
Syntax Analysis
Part I

Chapter 4

Position of a Parser in the Compiler Model



The Parser

- The task of the parser is to check syntax
- The syntax-directed translation stage in the compiler's front-end checks static semantics and produces an intermediate representation (IR) of the source program
  - Abstract syntax trees (ASTs)
  - Control-flow graphs (CFGs) with triples, three-address code, or register transfer lists
  - WHIRL (SGI Pro64 compiler) has 5 IR levels!

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 LL/LR parsers allows early detection of syntax errors
  - Goal: detection of an error as soon as possible without 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 is detected here

Prefix

```
...
DO 10 I = 1;0
...
```

Error Recovery Strategies

- *Panic mode*
  - Discard input until a token in a set of designated synchronizing tokens 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

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

Notational Conventions Used

- 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
  $L(G) = \{w \mid S \Rightarrow^+ w\}$

Derivation (Example)

$E \to E + E$

$E \to E * E$

$E \to (E)$

$E \to - E$

$E \to \mathbf{id}$

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

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

$E \Rightarrow^{*} E$

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

Chomsky Hierarchy: Language Classification

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

Chomsky Hierarchy

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

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

$L_1 = \{ \mathbf{a}^n\mathbf{b}^n \mid n \geq 1 \}$ is context free

$L_2 = \{ \mathbf{a}^n\mathbf{b}^n\mathbf{c}^n \mid n \geq 1 \}$ is context sensitive

Parsing

- *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

Top-Down Parsing

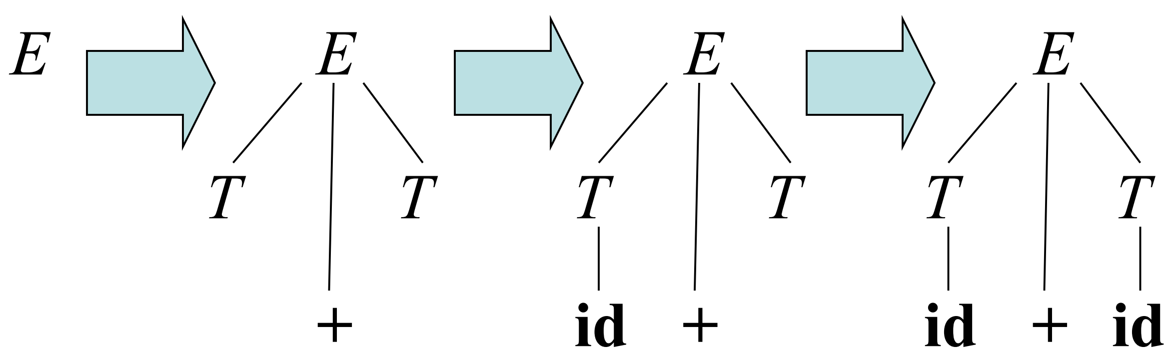
- LL methods (Left-to-right, Leftmost derivation) and recursive-descent parsing

Grammar:

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

Leftmost derivation:

$$E \Rightarrow_{lm} T + T$$
$$\Rightarrow_{lm} \mathbf{id} + T$$
$$\Rightarrow_{lm} \mathbf{id} + \mathbf{id}$$



Left Recursion (Recap)

- Productions of the form

$$A \to A\ \alpha \\ \quad\ \mid \beta \\ \quad\ \mid \gamma$$

  are left recursive
- When one of the productions in a grammar is left recursive then a predictive parser may loop forever

General Left Recursion Elimination

Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$

for $i = 1, \ldots, n$ **do**

  for $j = 1, \ldots, i\text{-}1$ **do**

    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

$$A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$$

  enddo

  eliminate the immediate left recursion in A_i

enddo

Immediate Left-Recursion Elimination

Rewrite every left-recursive production

$$
\begin{aligned}
A \rightarrow\ & A\ \alpha \\
\mid\ & \beta \\
\mid\ & \gamma \\
\mid\ & A\ \delta
\end{aligned}
$$

into a right-recursive production:

$$
\begin{aligned}
A \rightarrow\ & \beta\ A_R \\
\mid\ & \gamma\ A_R \\
A_R \rightarrow\ & \alpha\ A_R \\
\mid\ & \delta\ A_R \\
\mid\ & \varepsilon
\end{aligned}
$$

Example Left Rec. Elimination

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

$i = 1$: nothing to do

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

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

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

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

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

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

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

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

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

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

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

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

with

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

Predictive Parsing

- Eliminate left recursion from grammar
- Left factor the grammar
- Compute FIRST and FOLLOW
- Two variants:
  - Recursive (recursive calls)
  - Non-recursive (table-driven)

FIRST

- FIRST(α) = the set of terminals that begin all strings derived from α

  FIRST(a) = $\{a\}$ if $a \in T$
  FIRST(ε) = $\{\varepsilon\}$
  FIRST(A) = $\cup_{A\to\alpha}$ FIRST(α) for $A\to\alpha \in P$
  FIRST($X_1X_2\ldots X_k$) =
  if for all $j = 1, \ldots, i\text{-}1 : \varepsilon \in$ 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)$\setminus\{\varepsilon\}$ 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$)

LL(1) Grammar

- A grammar G is LL(1) if for each collections of productions
  $$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$
  for nonterminal A the following holds:

  1. $\mathrm{FIRST}(\alpha_i) \cap \mathrm{FIRST}(\alpha_j) = \varnothing$ for all $i \neq j$
  2. if $\alpha_i \Rightarrow^* \varepsilon$ then
     2.a. $\alpha_j \not\Rightarrow^* \varepsilon$ for all $i \neq j$
     2.b. $\mathrm{FIRST}(\alpha_j) \cap \mathrm{FOLLOW}(A) = \varnothing$
     for all $i \neq j$

Non-LL(1) Examples

Grammar	Not LL(1) because
$S \to S\ \mathbf{a} \mid \mathbf{a}$	Left recursive
$S \to \mathbf{a}\ S \mid \mathbf{a}$	FIRST($\mathbf{a}\ S$) $\cap$ FIRST($\mathbf{a}$) $\neq \emptyset$
$S \to \mathbf{a}\ R \mid \varepsilon$ $R \to S \mid \varepsilon$	For R: $S \to^* \varepsilon$ and $\varepsilon \to^* \varepsilon$
$S \to \mathbf{a}\ R\ \mathbf{a}$ $R \to S \mid \varepsilon$	For R: FIRST(S) $\cap$ FOLLOW(R) $\neq \emptyset$

Recursive Descent Parsing

- Grammar must be LL(1)
- Every nonterminal has one (recursive) procedure responsible for parsing the nonterminal's syntactic category of input tokens
- When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information

Using FIRST and FOLLOW to Write a Recursive Descent Parser

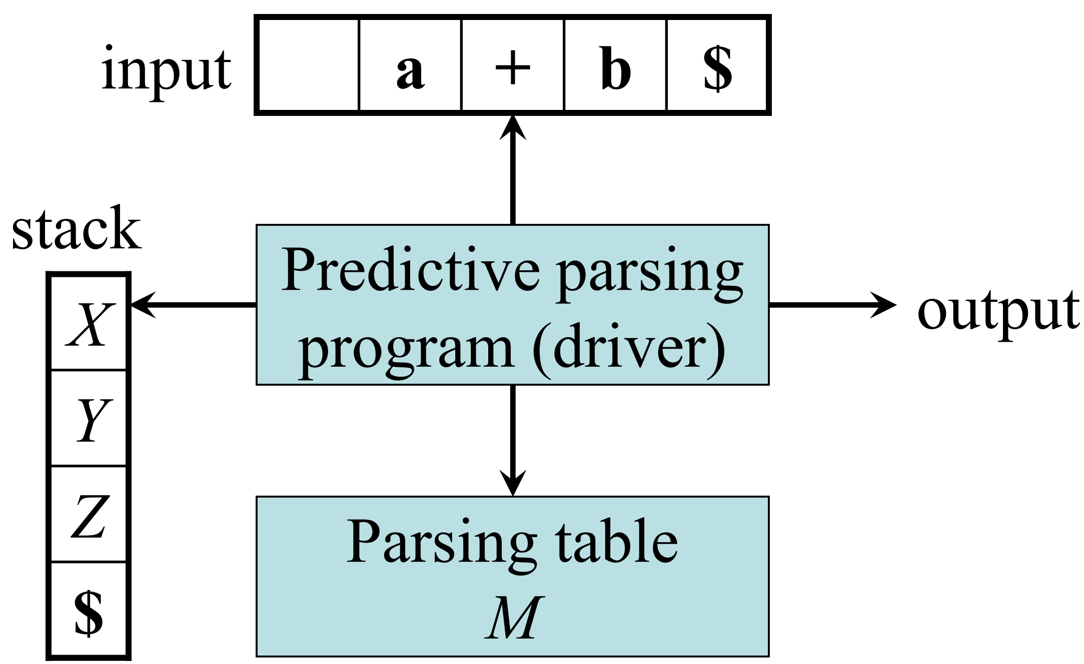
$$
\begin{aligned}
expr &\rightarrow term\ rest \\
rest &\rightarrow +\ term\ rest \\
&\quad \mid -\ term\ rest \\
&\quad \mid \varepsilon \\
term &\rightarrow \mathbf{id}
\end{aligned}
$$

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;

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

Non-Recursive Predictive 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 a Predictive Parsing Table

for each production $A \to \alpha$ **do**

  for each $a \in \text{FIRST}(\alpha)$ **do**

    add $A \to \alpha$ to $M[A,a]$

  enddo

  if $\varepsilon \in \text{FIRST}(\alpha)$ **then**

    for each $b \in \text{FOLLOW}(A)$ **do**

      add $A \to \alpha$ to $M[A,b]$

    enddo

  endif

enddo

Mark each undefined entry in M error

Example Table

$E \to T\ E_R$
$E_R \to +\ T\ E_R \mid \varepsilon$
$T \to F\ T_R$
$T_R \to *\ F\ T_R \mid \varepsilon$
$F \to (\ E\ ) \mid \mathbf{id}$

$A \to \alpha$	FIRST(α)	FOLLOW(A)
$E \to T\ E_R$	**(id**	**$)**
$E_R \to +\ T\ E_R$	+	**$)**
$E_R \to \varepsilon$	ε	
$T \to F\ T_R$	**(id**	**+ $)**
$T_R \to *\ F\ T_R$	*	**+ $)**
$T_R \to \varepsilon$	ε	
$F \to (\ E\ )$	**(**	*** + $)**
$F \to$ **id**	**id**	

	id	**+**	*****	**(**	**)**	**$**
E	$E \to T\ E_R$			$E \to T\ E_R$		
E_R		$E_R \to +\ T\ E_R$			$E_R \to \varepsilon$	$E_R \to \varepsilon$
T	$T \to F\ T_R$			$T \to F\ T_R$		
T_R		$T_R \to \varepsilon$	$T_R \to *\ F\ T_R$		$T_R \to \varepsilon$	$T_R \to \varepsilon$
F	$F \to$ **id**			$F \to (\ E\ )$		

LL(1) Grammars are Unambiguous

Ambiguous grammar
$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S_R \mid \mathbf{a}$
$S_R \rightarrow \mathbf{e}\ S \mid \varepsilon$
$E \rightarrow \mathbf{b}$

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S_R$	**i**	**e $**
$S \rightarrow \mathbf{a}$	**a**	
$S_R \rightarrow \mathbf{e}\ S$	**e**	**e $**
$S_R \rightarrow \varepsilon$	ε	
$E \rightarrow \mathbf{b}$	**b**	**t**

Error: duplicate table entry

	a	**b**	**e**	**i**	**t**	**$**
S	$S \rightarrow \mathbf{a}$			$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S_R$		
S_R			$S_R \rightarrow \varepsilon$ $S_R \rightarrow \mathbf{e}\ S$			$S_R \rightarrow \varepsilon$
E		$E \rightarrow \mathbf{b}$				

Predictive Parsing Program (Driver)

push(**$**)
push(S)
a := *lookahead*
repeat
    X := pop()
    if X is a terminal or X = **$ then**
        match(X) // move to next token, a := *lookahead*
    else if $M[X,a] = X \rightarrow Y_1 Y_2 \ldots Y_k$ **then**
        push(Y_k, $Y_{k\text{-}1}$, …, Y_2, Y_1) // such that Y_1 is on top
        produce output and/or invoke actions
    else   error()
    endif
until X = **$**

Example Table-Driven Parsing

Stack	Input	Production applied
$\$E$	**id+id*id$**	
$\$E_R T$	**id+id*id$**	$E \rightarrow T\ E_R$
$\$E_R T_R F$	**id+id*id$**	$T \rightarrow F\ T_R$
$\$E_R T_R\mathbf{id}$	**id+id*id$**	$F \rightarrow \mathbf{id}$
$\$E_R T_R$	**+id*id$**	
$\$E_R$	**+id*id$**	$T_R \rightarrow \varepsilon$
$\$E_R T+$	**+id*id$**	$E_R \rightarrow +\ T\ E_R$
$\$E_R T$	**id*id$**	
$\$E_R T_R F$	**id*id$**	$T \rightarrow F\ T_R$
$\$E_R T_R\mathbf{id}$	**id*id$**	$F \rightarrow \mathbf{id}$
$\$E_R T_R$	***id$**	
$\$E_R T_R F*$	***id$**	$T_R \rightarrow *\ F\ T_R$
$\$E_R T_R F$	**id$**	
$\$E_R T_R\mathbf{id}$	**id$**	$F \rightarrow \mathbf{id}$
$\$E_R T_R$	**$**	
$\$E_R$	**$**	$T_R \rightarrow \varepsilon$
$\$$	**$**	$E_R \rightarrow \varepsilon$

Panic Mode Recovery

Add synchronizing actions to undefined entries based on FOLLOW

FOLLOW(E) = { **$)** }
FOLLOW(E_R) = { **$)** }
FOLLOW(T) = { **+ $)** }
FOLLOW(T_R) = { **+ $)** }
FOLLOW(F) = { *** + $)** }

	id	+	*	**(**	**)**	**$**
E	$E \to T\, E_R$			$E \to T\, E_R$	***synch***	***synch***
E_R		$E_R \to + T\, E_R$			$E_R \to \varepsilon$	$E_R \to \varepsilon$
T	$T \to F\, T_R$	***synch***		$T \to F\, T_R$	***synch***	***synch***
T_R		$T_R \to \varepsilon$	$T_R \to * F\, T_R$		$T_R \to \varepsilon$	$T_R \to \varepsilon$
F	$F \to$ **id**	***synch***	***synch***	$F \to$ **(** E **)**	***synch***	***synch***

synch: pop A and skip input till synch token
or skip until FIRST(A) found

Phrase-Level Recovery

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

	id	+	*	**(**	**)**	**$**
E	$E \rightarrow T E_R$			$E \rightarrow T E_R$	***synch***	***synch***
E_R		$E_R \rightarrow + T E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F T_R$	***synch***		$T \rightarrow F T_R$	***synch***	***synch***
T_R	***insert ****	$T_R \rightarrow \varepsilon$	$T_R \rightarrow * F T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow$ **id**	***synch***	***synch***	$F \rightarrow$ **(** E **)**	***synch***	***synch***

insert *: insert missing * and redo the production

Error Productions

$E \to T\, E_R$
$E_R \to +\, T\, E_R \mid \varepsilon$
$T \to F\, T_R$
$T_R \to *\, F\, T_R \mid \varepsilon$
$F \to (\, E\,) \mid \mathbf{id}$

Add error production:

$$T_R \to F\, T_R$$

to ignore missing *****, e.g.: **id id**

	id	**+**	*****	**(**	**)**	**$**
E	$E \to T\, E_R$			$E \to T\, E_R$	***synch***	***synch***
E_R		$E_R \to +\, T\, E_R$			$E_R \to \varepsilon$	$E_R \to \varepsilon$
T	$T \to F\, T_R$	***synch***		$T \to F\, T_R$	***synch***	***synch***
T_R	$T_R \to F\, T_R$	$T_R \to \varepsilon$	$T_R \to *\, F\, T_R$		$T_R \to \varepsilon$	$T_R \to \varepsilon$
F	$F \to \mathbf{id}$	***synch***	***synch***	$F \to (\, E\,)$	***synch***	***synch***