Principles of Programming Languages

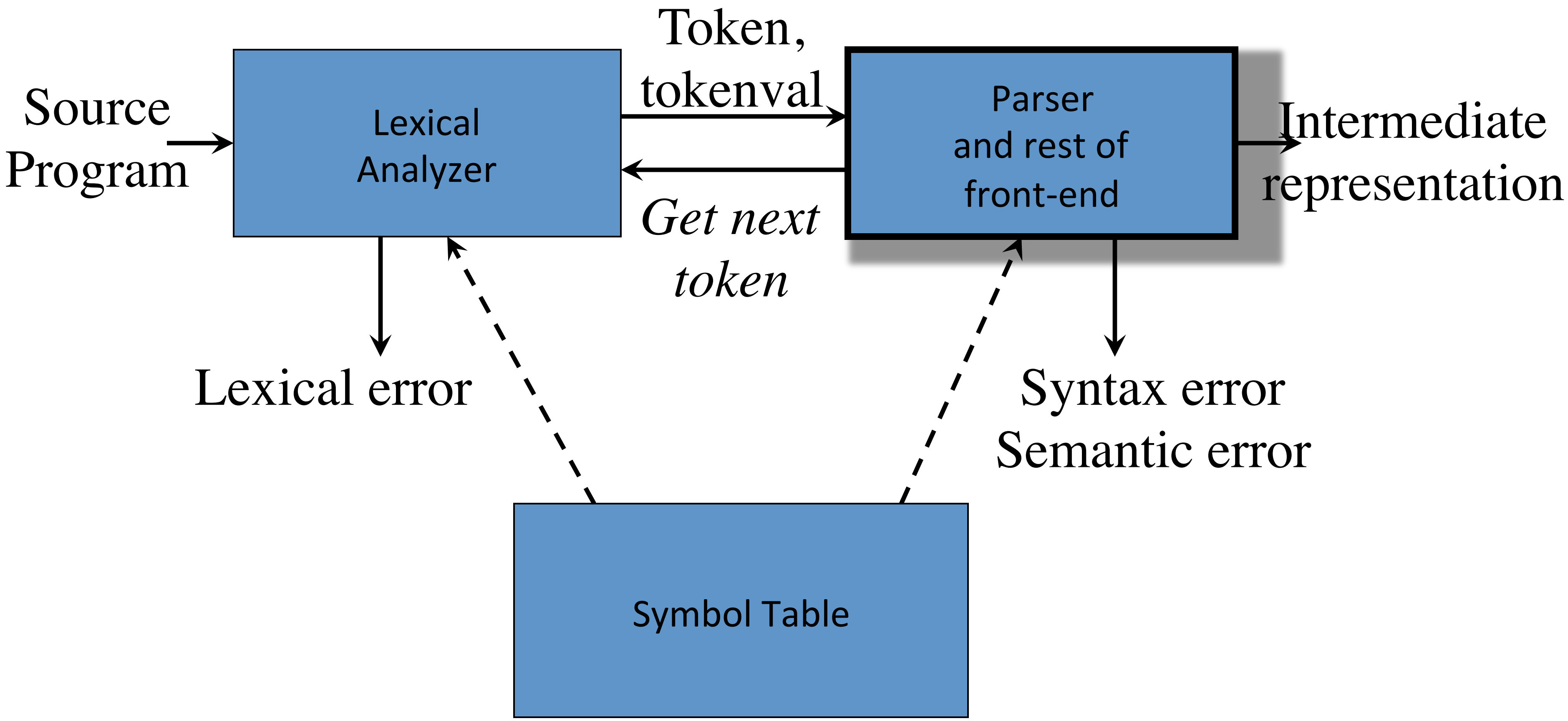
http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 7

- Parsing: general concepts
- Top-down parsing, LL(1) grammars
- Descent recursive
- Predictive non-recursive
- Error Handling

Position of a Parser in the Compiler Model



Source Program

Lexical Analyzer

Token, tokenval

Get next token

Parser and rest of front-end

Intermediate representation

Lexical error

Syntax error
Semantic error

Symbol Table

The Parser

- A parser implements a C-F grammar as a recognizer of strings
- The role of the parser in a compiler is twofold:
  1. To check syntax (= string recognizer)
     - And to report syntax errors accurately
  2. 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
- Now we focus on 1.

Error Handling

- A good compiler should assist in identifying and locating errors
  - *Lexical errors*: compiler can easily recover and continue (e.g. misspelled identifiers)
  - *Syntax errors*: can almost always recover (e.g. missing ‘;’ or ‘{‘, misplaced **case**)
  - *Static semantic errors*: can sometimes recover (e.g. type mismatches, variable used before declaration)
  - *Dynamic semantic errors*: hard or impossible to detect at compile time, runtime checks are required (e.g. null pointer, division by zero, invalid array access)
  - *Logical errors*: hard or impossible to detect (e.g. `if (b = true) …`)

Viable-Prefix Property

- The *viable-prefix property* of parsers allows early detection of syntax errors
  - Enjoyed by LL(1), LR(1) parsers
  - 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 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)

- A 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 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
  $$\gamma\ \alpha\ \delta \Rightarrow \gamma\ \beta\ \delta$$
  where $\alpha \rightarrow \beta$ is a production in the grammar
- In addition, we define
  - $\Rightarrow$ is *leftmost* $\Rightarrow_{lm}$ if γ does not contain a nonterminal
  - $\Rightarrow$ is *rightmost* $\Rightarrow_{rm}$ if δ 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}, {**+**,*****,**(**,**)**,**-**,**id**}, P, E) with productions

P = $E \rightarrow E \mathbf{+} E$

$E \rightarrow E \mathbf{*} E$

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

$E \rightarrow \mathbf{-}\ E$

$E \rightarrow \mathbf{id}$

Example derivations:

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

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

$E \Rightarrow^{*} E$

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

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

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

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

Where $\mathcal{L}(T) = \{\, L(G) \mid G \text{ is of type } T \,\}$
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

$L_2 = \{\, \mathbf{a}^n\mathbf{b}^n\mathbf{c}^n \mid n \geq 1 \,\}$ 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:

$E \rightarrow T \mathbf{+} T$

$T \rightarrow \mathbf{(}\ E\ \mathbf{)}$

$T \rightarrow \mathbf{-}\ E$

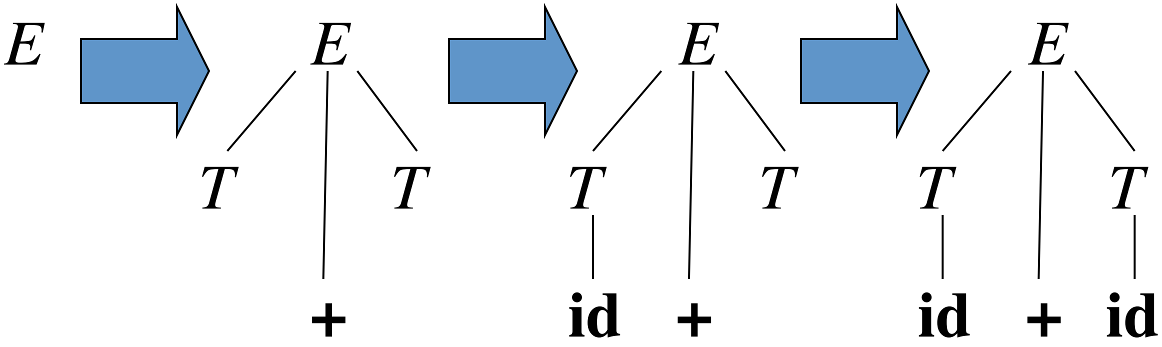
$T \rightarrow \mathbf{id}$

Leftmost derivation:

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

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

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



Left Recursion (Recap)

- Productions of the form
  $A \rightarrow A\,\alpha \mid \beta \mid \gamma$
  are left recursive
- Left recursion can be indirect
- When one of the productions in a grammar is (indirectly) left recursive, then a recursive descent or predictive parser loops forever on certain inputs

Immediate Left-Recursion Elimination

Rewrite every left-recursive production

$A \rightarrow A\ \alpha$
$\quad \mid \beta$
$\quad \mid \gamma$
$\quad \mid A\ \delta$

into a right-recursive production:

$A \rightarrow \beta\ A_R$
$\quad \mid \gamma\ A_R$
$A_R \rightarrow \alpha\ A_R$
$\quad \mid \delta\ A_R$
$\quad \mid \varepsilon$

A General Systematic Left Recursion Elimination Method

```
Input: Grammar G with no cycles or ε-productions
Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$
for $i = 1, \ldots, n$ do
    for $j = 1, \ldots, i-1$ 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
```

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

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

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

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

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

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

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

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

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

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

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

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

  $A \rightarrow \alpha\, A_R \mid \gamma$

  $A_R \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

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)

FIRST (Revisited)

- FIRST(α) = { *the set of terminals that begin all strings derived from* α }
- FIRST(a) = {a}   if $a \in T$
- FIRST(ε) = {ε}
- FIRST(A) = $\cup_{A\rightarrow\alpha}$ FIRST(α)   for $A \rightarrow \alpha \in P$
- FIRST($X_1X_2\ldots X_k$) =

  if for all j = 1, …, 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, …, 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 ($B \rightarrow \alpha A \beta$) $\in P$ do
    add FIRST($\beta$)\{$\varepsilon$} to FOLLOW($A$)
for all ($B \rightarrow \alpha A \beta$) $\in P$ and $\varepsilon \in$ FIRST($\beta$) do
    add FOLLOW($B$) to FOLLOW($A$)
for all ($B \rightarrow \alpha A$) $\in P$ 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 it is not left recursive and for each collection of productions
  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
  for nonterminal A the following holds:

  1. $\mathrm{FIRST}(\alpha_i) \cap \mathrm{FIRST}(\alpha_j) = \emptyset$ 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. $\mathrm{FIRST}(\alpha_j) \cap \mathrm{FOLLOW}(A) = \emptyset$ for all $i \neq j$

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($\mathbf{a}$) $\neq \emptyset$
$S \rightarrow \mathbf{a}\ R \mid \varepsilon$ $R \rightarrow S \mid \varepsilon$	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) $\neq \emptyset$

Recursive-Descent Parsing (Recap)

- 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 in a Recursive-Descent Parser

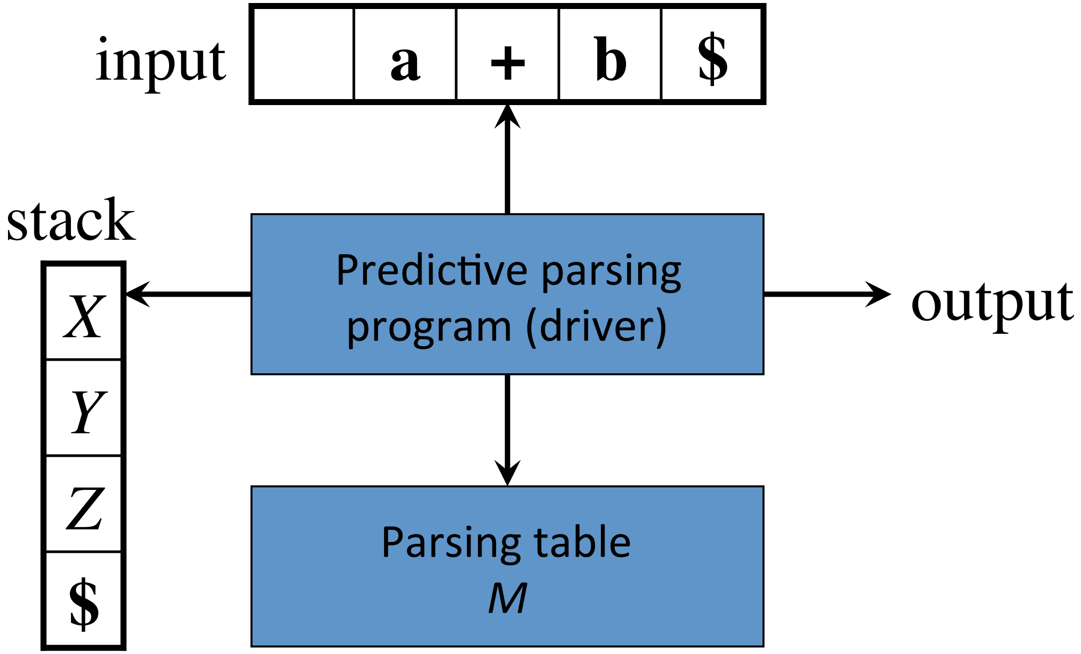
expr → *term rest*
rest → **+** *term rest*
| **-** *term rest*
| ε
term → **id**

```
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 $A \rightarrow \alpha$ do
    for each $a \in$ FIRST($\alpha$) do
        add production $A \rightarrow \alpha$ to $M[A,a]$
    enddo
    if $\varepsilon \in$ FIRST($\alpha$) then
        for each $b \in$ FOLLOW($A$) do
            add $A \rightarrow \alpha$ to $M[A,b]$
        enddo
    endif
enddo
Mark each undefined entry in $M$ error
```

Example Table

$E \rightarrow T\ E_R$

$E_R \rightarrow \mathbf{+}\ T\ E_R \mid \varepsilon$

$T \rightarrow F\ T_R$

$T_R \rightarrow *\ F\ T_R \mid \varepsilon$

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

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$E \rightarrow T\ E_R$	**(id**	**$)**
$E_R \rightarrow \mathbf{+}\ T\ E_R$	**+**	**$)**
$E_R \rightarrow \varepsilon$	ε	
$T \rightarrow F\ T_R$	**(id**	**+ $)**
$T_R \rightarrow *\ F\ T_R$	*	**+ $)**
$T_R \rightarrow \varepsilon$	ε	
$F \rightarrow (\ E\ )$	(	* **+ $)**
$F \rightarrow \mathbf{id}$	**id**	* **+ $)**

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

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($X$) // moves to next token and $a$ := lookahead
    else if $M[X,a] = X \to Y_1 Y_2 \ldots Y_k$ then
        push($Y_k, Y_{k-1}, \ldots, Y_2, Y_1$) // such that $Y_1$ is on top
        … invoke actions and/or produce IR output …
    else    error()
    endif
until $X$ = $
```

Example Table-Driven Parsing

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

Panic Mode Recovery

Add synchronizing actions to undefined entries based on FOLLOW

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

Pro: Can be automated
Cons: Error messages are needed

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

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	$E \rightarrow T\, E_R$			$E \rightarrow T\, E_R$	***synch***	***synch***
E_R		$E_R \rightarrow \mathbf{+}\, T\, E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
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 *: driver inserts missing * and retries the production

Error Productions

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

Add "*error production*":

$T_R \rightarrow F\ T_R$

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

Pro: Powerful recovery method
Cons: Manual addition of productions

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