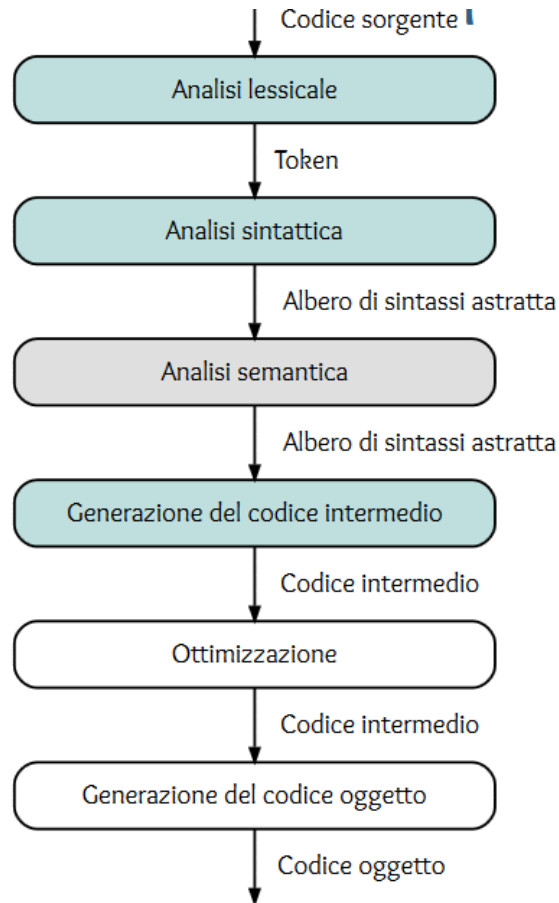


Laboratorio di Linguaggi Formali e Traduttori

LFT lab T2, a.a. 2022/2023

Analisi sintattica

Analisi sintattica



- **Analisi sintattica:**
 - Fase successiva a quella dell'analisi lessicale.
 - Input: la sequenza di token (l'output dell'analisi lessicale) che corrisponde al programma in input.
 - Se il programma in input corrisponde alla grammatica del linguaggio: costruisce un albero di sintattico/produce una derivazione.
 - Se il programma in input *non* corrisponde alla grammatica del linguaggio: fa output di un messaggio di errore.
- **Questa lezione:**
 - Realizzare un parser (riconoscitore) a discesa ricorsiva per una grammatiche LL(1) per la generazione di espressioni aritmetiche semplici.

Parsing a discesa ricorsiva

- Una procedura associata ad ogni variabile.
- Procedura di una variabile: codice derivato dalle produzioni associate con la variabile.
- Dato un input da analizzare, l'albero di chiamate delle procedure ricorsive corrisponde all'albero sintattico.

$\langle start \rangle$	$::=$	$\langle expr \rangle EOF$
$\langle expr \rangle$	$::=$	$\langle term \rangle \langle exprp \rangle$
$\langle exprp \rangle$	$::=$	$+ \langle term \rangle \langle exprp \rangle$ $ - \langle term \rangle \langle exprp \rangle$ $ \varepsilon$
$\langle term \rangle$	$::=$	$\langle fact \rangle \langle termp \rangle$
$\langle termp \rangle$	$::=$	$* \langle fact \rangle \langle termp \rangle$ $ / \langle fact \rangle \langle termp \rangle$ $ \varepsilon$
$\langle fact \rangle$	$::=$	$(\langle expr \rangle) \mid NUM$

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}  
  
private void expr() {  
    // ... completare ...  
}  
  
private void exprp() {  
    switch (look.tag) {  
        case '+':  
            // ... completare ...  
    }  
}  
  
private void term() {  
    // ... completare ...  
}  
  
private void termp() {  
    // ... completare ...  
}  
  
private void fact() {  
    // ... completare ...  
}
```

Parsing a discesa ricorsiva

- Una procedura associata ad ogni variabile.
- Procedura di una variabile: codice derivato dalle produzioni associate con la variabile.
- Dato un input da analizzare, l'albero di chiamate delle procedure ricorsive corrisponde all'albero sintattico.

$\langle start \rangle$	$::=$	$\langle expr \rangle EOF$
$\langle expr \rangle$	$::=$	$\langle term \rangle \langle exprp \rangle$
$\langle exprp \rangle$	$::=$	$+ \langle term \rangle \langle exprp \rangle$ $ - \langle term \rangle \langle exprp \rangle$ $ \varepsilon$
$\langle term \rangle$	$::=$	$\langle fact \rangle \langle termp \rangle$
$\langle termp \rangle$	$::=$	$* \langle fact \rangle \langle termp \rangle$ $ / \langle fact \rangle \langle termp \rangle$ $ \varepsilon$
$\langle fact \rangle$	$::=$	$(\langle expr \rangle) \mid NUM$

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}
```

```
private void expr() {  
    // ... completare ...  
}
```

```
private void exprp() {  
    switch (look.tag) {  
        case '+':  
            // ... completare ...  
    }  
}
```

```
private void term() {  
    // ... completare ...  
}
```

```
private void termp() {  
    // ... completare ...  
}
```

```
private void fact() {  
    // ... completare ...  
}
```

Parsing a discesa ricorsiva

- Una procedura associata ad ogni variabile.
- Procedura di una variabile: codice derivato dalle produzioni associate con la variabile.
- Dato un input da analizzare, l'albero di chiamate delle procedure ricorsive corrisponde all'albero sintattico.

$\langle start \rangle$	$::=$	$\langle expr \rangle EOF$
$\langle expr \rangle$	$::=$	$\langle term \rangle \langle exprp \rangle$
$\langle exprp \rangle$	$::=$	$+ \langle term \rangle \langle exprp \rangle$ $ - \langle term \rangle \langle exprp \rangle$ $ \varepsilon$
$\langle term \rangle$	$::=$	$\langle fact \rangle \langle termp \rangle$
$\langle termp \rangle$	$::=$	$* \langle fact \rangle \langle termp \rangle$ $ / \langle fact \rangle \langle termp \rangle$ $ \varepsilon$
$\langle fact \rangle$	$::=$	$(\langle expr \rangle) \mid NUM$

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}
```

```
private void expr() {  
    // ... completare ...  
}
```

```
private void exprp() {  
    switch (look.tag) {  
        case '+':  
            // ... completare ...  
    }  
}
```

```
private void term() {  
    // ... completare ...  
}
```

```
private void termp() {  
    // ... completare ...  
}
```

```
private void fact() {  
    // ... completare ...  
}
```

Parsing a discesa ricorsiva

- Una procedura associata ad ogni variabile.
- Procedura di una variabile: codice derivato dalle produzioni associate con la variabile.
- Dato un input da analizzare, l'albero di chiamate delle procedure ricorsive corrisponde all'albero sintattico.

$\langle start \rangle$	$::=$	$\langle expr \rangle EOF$
$\langle expr \rangle$	$::=$	$\langle term \rangle \langle exprp \rangle$
$\langle exprp \rangle$	$::=$	$+ \langle term \rangle \langle exprp \rangle$ $- \langle term \rangle \langle exprp \rangle$ ε
$\langle term \rangle$	$::=$	$\langle fact \rangle \langle termp \rangle$
$\langle termp \rangle$	$::=$	$* \langle fact \rangle \langle termp \rangle$ $/ \langle fact \rangle \langle termp \rangle$ ε
$\langle fact \rangle$	$::=$	$(\langle expr \rangle) \mid NUM$

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}
```

```
private void expr() {  
    // ... completare ...  
}
```

```
private void exprp() {  
    switch (look.tag) {  
        case '+':  
            // ... completare ...  
    }  
}
```

```
private void term() {  
    // ... completare ...  
}
```

```
private void termp() {  
    // ... completare ...  
}
```

```
private void fact() {  
    // ... completare ...  
}
```

Parsing a discesa ricorsiva

- Una procedura associata ad ogni variabile.
- Procedura di una variabile: codice derivato dalle produzioni associate con la variabile.
- Dato un input da analizzare, l'albero di chiamate delle procedure ricorsive corrisponde all'albero sintattico.

$\langle start \rangle$	$::=$	$\langle expr \rangle EOF$
$\langle expr \rangle$	$::=$	$\langle term \rangle \langle exprp \rangle$
$\langle exprp \rangle$	$::=$	$+ \langle term \rangle \langle exprp \rangle$ $ - \langle term \rangle \langle exprp \rangle$ $ \varepsilon$
$\langle term \rangle$	$::=$	$\langle fact \rangle \langle termp \rangle$
$\langle termp \rangle$	$::=$	$* \langle fact \rangle \langle termp \rangle$ $ / \langle fact \rangle \langle termp \rangle$ $ \varepsilon$
$\langle fact \rangle$	$::=$	$(\langle expr \rangle) \mid NUM$

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}  
  
private void expr() {  
    // ... completare ...  
}  
  
private void exprp() {  
    switch (look.tag) {  
        case '+':  
            // ... completare ...  
    }  
}  
  
private void term() {  
    // ... completare ...  
}  
  
private void termp() {  
    // ... completare ...  
}  
  
private void fact() {  
    // ... completare ...  
}
```

Parsing a discesa ricorsiva

- Una procedura associata ad ogni variabile.
- Procedura di una variabile: codice derivato dalle produzioni associate con la variabile.
- Dato un input da analizzare, l'albero di chiamate delle procedure ricorsive corrisponde all'albero sintattico.

$\langle start \rangle$	$::=$	$\langle expr \rangle EOF$
$\langle expr \rangle$	$::=$	$\langle term \rangle \langle exprp \rangle$
$\langle exprp \rangle$	$::=$	$+ \langle term \rangle \langle exprp \rangle$ $ - \langle term \rangle \langle exprp \rangle$ $ \varepsilon$
$\langle term \rangle$	$::=$	$\langle fact \rangle \langle termp \rangle$
$\langle termp \rangle$	$::=$	$* \langle fact \rangle \langle termp \rangle$ $ / \langle fact \rangle \langle termp \rangle$ $ \varepsilon$
$\langle fact \rangle$	$::=$	$(\langle expr \rangle) \mid NUM$

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}  
  
private void expr() {  
    // ... completare ...  
}  
  
private void exprp() {  
    switch (look.tag) {  
        case '+':  
            // ... completare ...  
    }  
}  
  
private void term() {  
    // ... completare ...  
}  
  
private void termp() {  
    // ... completare ...  
}  
  
private void fact() {  
    // ... completare ...  
}
```


Parsing a discesa ricorsiva

- Una procedura associata ad ogni variabile.
- Procedura di una variabile: codice derivato dalle produzioni associate con la variabile.
- Dato un input da analizzare, l'albero di chiamate delle procedure ricorsive corrisponde all'albero sintattico.

$\langle start \rangle$	$::=$	$\langle expr \rangle EOF$
$\langle expr \rangle$	$::=$	$\langle term \rangle \langle exprp \rangle$
$\langle exprp \rangle$	$::=$	$+ \langle term \rangle \langle exprp \rangle$ $ - \langle term \rangle \langle exprp \rangle$ $ \varepsilon$
$\langle term \rangle$	$::=$	$\langle fact \rangle \langle termp \rangle$
$\langle termp \rangle$	$::=$	$* \langle fact \rangle \langle termp \rangle$ $ / \langle fact \rangle \langle termp \rangle$ $ \varepsilon$
$\langle fact \rangle$	$::=$	$(\langle expr \rangle) NUM$

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}  
  
private void expr() {  
    // ... completare ...  
}  
  
private void exprp() {  
    switch (look.tag) {  
        case '+':  
            // ... completare ...  
    }  
}  
  
private void term() {  
    // ... completare ...  
}  
  
private void termp() {  
    // ... completare ...  
}  
  
private void fact() {  
    // ... completare ...  
}
```

Parsing a discesa ricorsiva

- Esempio: variabile $\langle start \rangle$.
- Grammatica:
 - Una singola produzione è associata con $\langle start \rangle$.
 - La produzione consiste di una variabile $\langle expr \rangle$, seguito da un simbolo terminale EOF.

$\langle start \rangle ::= \langle expr \rangle \text{ EOF}$

- Codice:
 - Una chiamata alla procedura `expr...`
 - ... seguita da un controllo rispetto a EOF.

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}
```

Parsing a discesa ricorsiva

- Esempio: variabile $\langle start \rangle$.
- Grammatica:
 - Una singola produzione è associata con $\langle start \rangle$.
 - La produzione consiste di una variabile $\langle expr \rangle$, seguito da un simbolo terminale EOF.

$\langle start \rangle$	$::=$	$\langle expr \rangle$	EOF
-------------------------	-------	------------------------	-----

- Codice:

- Una chiamata alla procedura `expr...`
- ... seguita da un controllo rispetto a EOF.

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}
```

Parsing a discesa ricorsiva

- Esempio: variabile $\langle start \rangle$.
- Grammatica:
 - Una singola produzione è associata con $\langle start \rangle$.
 - La produzione consiste di una variabile $\langle expr \rangle$, seguito da un simbolo terminale EOF.

$\langle start \rangle ::= \langle expr \rangle \text{ EOF}$

- Codice:
 - Una chiamata alla procedura `expr...`
 - ... seguita da un controllo rispetto a EOF.

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}
```

Parsing a discesa ricorsiva

- Pseudo-codice per la parsificazione a discesa ricorsiva (slide 4 del file «parsing_rd.pdf» con titolo «4.2 Parsing ricorsivo discendente»).

```
var  $w$  : string                                     //  $w$  è la stringa da riconoscere con $ in fondo
var  $i$  : int                                           //  $i$  è l'indice del prossimo simbolo di  $w$  da leggere

procedure match( $a$  : symbol)
  if  $w[i] = a$  then  $i \leftarrow i + 1$  else error

procedure parse( $v$  : string)                          //  $v$  è la stringa da riconoscere
   $w \leftarrow v\$$ 
   $i \leftarrow 0$ 
   $S()$                                                 //  $S$  è il simbolo iniziale della grammatica
  match( $\$$ )                                           // controlla di aver letto tutta la stringa

procedure  $A()$                                        //  $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$  sono le produzioni per  $A$ 
  if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_1)$  then
    :
  else if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_k)$  then
    for  $X \in \alpha_k$  do
      if  $X$  è un terminale then match( $X$ ) else  $X()$ 
    :
  else error                                          //  $w[i]$  non è nell'insieme guida di nessuna produzione per  $A$ 
```

Codice: classe Parser

```
var w : string           // w è la stringa da riconoscere con $ in fondo
var i : int              // i è l'indice del prossimo simbolo di w da leggere

procedure match(a : symbol)
  if w[i] = a then i ← i + 1 else error

procedure parse(v : string)           // v è la stringa da riconoscere
  w ← v$
  i ← 0
  S()                                // S è il simbolo iniziale della grammatica
  match($)                           // controlla di aver letto tutta la stringa

procedure A()                         // A → α1 | ... | αn sono le produzioni per A
  if w[i] ∈ GUIDA(A → α1) then
  :
  else if w[i] ∈ GUIDA(A → αk) then
    for X ∈ αk do
      if X è un terminale then match(X) else X()
  :
  else error                        // w[i] non è nell'insieme guida di nessuna produzione per A
```

```
void move() {
  look = lex.lexical_scan(pbr);
  System.out.println("token = " + look);
}

void error(String s) {
  throw new Error("near line " + lex.line + ": " + s);
}

void match(int t) {
  if (look.tag == t) {
    if (look.tag != Tag.EOF) move();
  } else error("syntax error");
}
```

Codice: classe Parser

```
var w : string
var i : int

// w è la stringa da riconoscere con $ in fondo
// i è l'indice del prossimo simbolo di w da leggere

procedure match(a : symbol)
  if w[i] = a then i ← i + 1 else error

procedure parse(v : string)
  w ← v$
  i ← 0
  S()
  match($)
  // v è la stringa da riconoscere
  // S è il simbolo iniziale della grammatica
  // controlla di aver letto tutta la stringa

procedure A()
  if w[i] ∈ GUIDA(A → α1) then
    :
  else if w[i] ∈ GUIDA(A → αk) then
    for X ∈ αk do
      if X è un terminale then match(X) else X()
    :
  else error
  // w[i] non è nell'insieme guida di nessuna produzione per A
```

```
void move() {
    look = lex.lexical_scan(pbr);
    System.out.println("token = " + look);
}

void error(String s) {
    throw new Error("near line " + lex.line + ": " + s);
}

void match(int t) {
    if (look.tag == t) {
        if (look.tag != Tag.EOF) move();
    } else error("syntax error");
}
```

Pseudo-codice	Codice
w[i]	look
a	t
i ← i+1	move()
error	error()

Codice: classe Parser

```
var w : string
var i : int

// w è la stringa da riconoscere con $ in fondo
// i è l'indice del prossimo simbolo di w da leggere

procedure match(a : symbol)
  if w[i] = a then i ← i + 1 else error

procedure parse(v : string)
  w ← v$
  i ← 0
  S()
  match($)
  // v è la stringa da riconoscere
  // S è il simbolo iniziale della grammatica
  // controlla di aver letto tutta la stringa

procedure A()
  if w[i] ∈ GUIDA(A → α1) then
    :
  else if w[i] ∈ GUIDA(A → αk) then
    for X ∈ αk do
      if X è un terminale then match(X) else X()
    :
  else error
  // A → α1 | ... | αn sono le produzioni per A
  // w[i] non è nell'insieme guida di nessuna produzione per A
```

```
void move() {
    look = lex.lexical_scan(pbr);
    System.out.println("token = " + look);
}

void error(String s) {
    throw new Error("near line " + lex.line + ": " + s);
}

void match(int t) {
    if (look.tag == t) {
        if (look.tag != Tag.EOF) move();
    } else error("syntax error");
}
```

Pseudo-codice	Codice
w[i]	look
a	t
i ← i+1	move()
error	error()

Codice: classe Parser

```
procedure A()                                     //  $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$  sono le produzioni per  $A$ 
  if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_1)$  then
  :
  else if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_k)$  then
    for  $X \in \alpha_k$  do
      if  $X$  è un terminale then match( $X$ ) else  $X()$ 
    :
  else error                                       //  $w[i]$  non è nell'insieme guida di nessuna produzione per  $A$ 
```

$\langle start \rangle ::= \langle expr \rangle \text{ EOF}$

Variabile $\langle start \rangle$ ha una sola produzione

```
public void start() {
    // ... completare ...
    expr();
    match(Tag.EOF);
    // ... completare ...
}
```

Codice: classe Parser

```
procedure A()  
  if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_1)$  then  
    .  
  else if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_k)$  then  
    for  $X \in \alpha_k$  do  
      if  $X$  è un terminale then match( $X$ ) else  $X()$   
    .  
  else error
```

// $A \rightarrow \alpha_1 \mid \dots \mid \alpha_k$ sono le produzioni per A

// $w[i]$ non è nell'insieme guida di nessuna produzione per A

$\langle start \rangle ::= \langle expr \rangle \text{ EOF}$

Variabile $\langle start \rangle$ ha una sola produzione

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}
```

Codice: classe Parser

```
procedure A()  
  if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_1)$  then  
    ...  
  else if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_k)$  then  
    for  $X \in \alpha_k$  do  
      if  $X$  è un terminale then match( $X$ ) else  $X()$   
    ...  
  else error
```

// $A \rightarrow \alpha_1 \mid \dots \mid \alpha_k$ sono le produzioni per A

// $w[i]$ non è nell'insieme guida di nessuna produzione per A

$\langle start \rangle ::= \langle expr \rangle \text{ EOF}$

Variabile $\langle start \rangle$ ha una sola produzione

```
public void start() {  
  // ... completare ...  
  expr();  
  match(Tag.EOF);  
  // ... completare ...  
}
```

Codice: classe Parser

```
procedure A()  
  if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_1)$  then  
    .  
  else if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_k)$  then  
    for  $X \in \alpha_k$  do  
      if  $X$  è un terminale then match( $X$ ) else  $X()$   
    .  
  else error
```

// $A \rightarrow \alpha_1 \mid \dots \mid \alpha_k$ sono le produzioni per A

// $w[i]$ non è nell'insieme guida di nessuna produzione per A

$\langle start \rangle ::= \langle expr \rangle \text{ EOF}$

Variabile $\langle start \rangle$ ha una sola produzione

```
public void start() {  
  // ... completare ...  
  expr();  
  match(Tag.EOF);  
  // ... completare ...  
}
```

Codice: classe Parser

```
procedure A()  
  if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_1)$  then  
    .  
  else if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_k)$  then  
    for  $X \in \alpha_k$  do  
      if  $X$  è un terminale then match( $X$ ) else  $X()$   
    .  
  else error
```

$// A \rightarrow \alpha_1 | \dots | \alpha_n$ sono le produzioni per A

$// w[i]$ non è nell'insieme guida di nessuna produzione per A

$\langle start \rangle$	$::=$	$\langle expr \rangle$	EOF
-------------------------	-------	------------------------	-----

Variabile $\langle start \rangle$ ha una sola produzione

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}
```

Codice: classe Parser

```
procedure A()  
  if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_1)$  then  
    .  
  else if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_k)$  then  
    for  $X \in \alpha_k$  do  
      if  $X$  è un terminale then match(X) else  $X()$   
    .  
  else error
```

α_k
 $// A \rightarrow \alpha_1 | \dots | \alpha_n$ sono le produzioni per A

$// w[i]$ non è nell'insieme guida di nessuna produzione per A

$\langle start \rangle ::= \langle expr \rangle$ EOF

Variabile $\langle start \rangle$ ha una sola produzione

```
public void start() {  
    // ... completare ...  
    expr();  
    match(Tag.EOF);  
    // ... completare ...  
}
```

Codice: classe Parser

```
procedure A()                                     //  $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$  sono le produzioni per  $A$ 
  if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_1)$  then
  :
  else if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_k)$  then
    for  $X \in \alpha_k$  do
      if  $X$  è un terminale then match( $X$ ) else  $X()$ 
    :
  else error                                     //  $w[i]$  non è nell'insieme guida di nessuna produzione per  $A$ 
```

$\langle \text{exprp} \rangle$	$::=$	$+$	$\langle \text{term} \rangle$	$\langle \text{exprp} \rangle$	
			$-$	$\langle \text{term} \rangle$	$\langle \text{exprp} \rangle$
			ε		

Variabile $\langle \text{exprp} \rangle$ ha più produzioni

```
private void exprp() {
  switch (look.tag) {
  case '+':
    // ... completare ...
  }
}
```

Codice: classe Parser

```
procedure A()  
  if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_1)$  then  
    :  
  else if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_k)$  then  
    for  $X \in \alpha_k$  do  
      if  $X$  è un terminale then match( $X$ ) else  $X()$   
    :  
  else error
```

$\alpha_1 | \alpha_2 | \alpha_3$
// $A \rightarrow \alpha_1 | \alpha_2 | \alpha_3$ sono le produzioni per A

// $w[i]$ non è nell'insieme guida di nessuna produzione per A

$\langle \text{exprp} \rangle$	$::=$	$+ \langle \text{term} \rangle \langle \text{exprp} \rangle$	α_1
		$- \langle \text{term} \rangle \langle \text{exprp} \rangle$	α_2
		ε	α_3

Variabile $\langle \text{exprp} \rangle$ ha più produzioni

```
private void exprp() {  
  switch (look.tag) {  
    case '+':  
      // ... completare ...  
  }  
}
```


Codice: classe Parser

```
procedure A()  
  if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_1)$  then  
    :  
  else if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_k)$  then  
    for  $X \in \alpha_k$  do  
      if  $X$  è un terminale then match( $X$ ) else  $X()$   
    :  
  else error
```

$\alpha_1 | \alpha_2 | \alpha_3$
// $A \rightarrow \alpha_1 | \alpha_2 | \alpha_3$ sono le produzioni per A

// $w[i]$ non è nell'insieme guida di nessuna produzione per A

$\langle \text{exprp} \rangle$	$::=$	$+ \langle \text{term} \rangle \langle \text{exprp} \rangle$	α_1
		$- \langle \text{term} \rangle \langle \text{exprp} \rangle$	α_2
		ϵ	α_3

Variabile $\langle \text{exprp} \rangle$ ha più produzioni

```
private void exprp() {  
  switch (look.tag) {  
    case '+':  
      // ... completare ...  
  }  
}
```

Codice: classe Parser

```
procedure A()  
  if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_1)$  then  
    :  
  else if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_k)$  then  
    for  $X \in \alpha_k$  do  
      if  $X$  è un terminale then match( $X$ ) else  $X()$   
    :  
  else error  
//  $w[i]$  non è nell'insieme guida di nessuna produzione per  $A$ 
```

$\alpha_1 | \alpha_2 | \alpha_3$ sono le produzioni per A

$\langle \text{exprp} \rangle$	$::=$	$+ \langle \text{term} \rangle \langle \text{exprp} \rangle$	α_1
		$- \langle \text{term} \rangle \langle \text{exprp} \rangle$	α_2
		ϵ	α_3

Variabile $\langle \text{exprp} \rangle$ ha più produzioni

```
private void exprp() {  
  switch (look.tag) {  
    case '+':  
      // ... completare ...  
  }  
}
```

Messaggi di errore

```
void error(String s) {  
    throw new Error("near line " + lex.line + ": " + s);  
}
```

- Argomento `s`: utilizzare per dare informazione utile all'utente del programma quando l'input non corrisponde alla grammatica.
- Consiglio: segnalare la procedura che invoca `error` (ad esempio, `error("Error in term")`).
- Tutte le procedure devono avere dei meccanismi per rilevare input erraneo, in modo tale che eventuali errori sono segnalati appena possibile.

Input	Procedura in cui è rilevato l'errore
) 2	start
2+(expr
5+)	term
1+2(termp
1*+2	fact
1+2)	match (in questo caso il messaggio di errore è semplicemente "syntax error")

Codice: classe Parser

```
procedure parse(v : string)                                // v è la stringa da riconoscere
  w ← v$
  i ← 0
  S()                                                       // S è il simbolo iniziale della grammatica
  match($)                                                   // controlla di aver letto tutta la stringa
```

- `main`: simile a `parse`; entrambe chiamano la procedura della variabile iniziale della grammatica (`start`).
- Si nota che la fine del input è rappresentato esplicitamente con il terminale EOF quindi, in nostro caso, non c'è la necessità di controllare '\$' nel `main` (è controllato nella procedura `start` con `match(Tag.EOF)`).

```
public static void main(String[] args) {
    Lexer lex = new Lexer();
    String path = "...path..."; // il percorso del file da leggere
    try {
        BufferedReader br = new BufferedReader(new FileReader(path));
        Parser parser = new Parser(lex, br);
        parser.start();
        System.out.println("Input OK");
        br.close();
    } catch (IOException e) {e.printStackTrace();}
}
```