

# Laboratorio di Linguaggi Formali e Traduttori

## LFT lab T4, a.a. 2022/2023

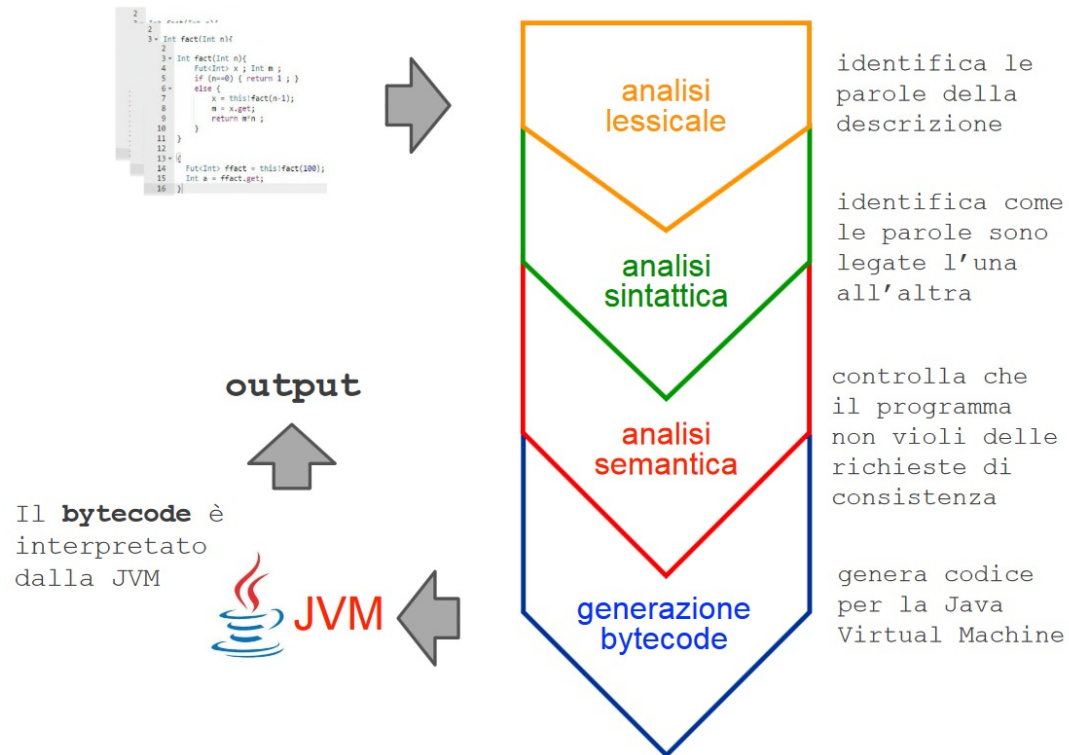
Docente: Luigi Di Caro

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 parsificazione/produce una derivazione.
- Se il programma in input non corrisponde alla grammatica del linguaggio: fa output di un messaggio di errore.



# Analisi sintattica

- Obiettivo dell'esercizio 3.1: scrivere un programma per l'analisi sintattica per **espressioni aritmetiche** semplici, scritte in notazione infissa.
  - Elementi: numeri non negativi; operatori di somma, sottrazione, moltiplicazione e divisione; simboli di parentesi tonda ( e ).

## corrispondenza tra notazioni

*parentesi angolari*  $\rightarrow$  *variabile*  
 $::=$   $\rightarrow$  *produzione*  
 $+ - * / ( ) \text{ NUM}$   $\rightarrow$  *simboli terminali*  
 $\epsilon$   $\rightarrow$  *stringa vuota*

|                         |       |  |
|-------------------------|-------|--|
| $\langle start \rangle$ | $::=$ | $\langle expr \rangle \text{ EOF}$   |
| $\langle expr \rangle$  | $::=$ | $\langle term \rangle \langle exprp \rangle$   |
| $\langle exprp \rangle$ | $::=$ | $+ \langle term \rangle \langle exprp \rangle$<br>$ $<br>$- \langle term \rangle \langle exprp \rangle$<br>$ $<br>$\epsilon$ |
| $\langle term \rangle$  | $::=$ | $\langle fact \rangle \langle termp \rangle$   |
| $\langle termp \rangle$ | $::=$ | $* \langle fact \rangle \langle termp \rangle$<br>$ $<br>$/ \langle fact \rangle \langle termp \rangle$<br>$ $<br>$\epsilon$ |
| $\langle fact \rangle$  | $::=$ | $( \langle expr \rangle ) \mid \text{ NUM}$  |

# Analisi sintattica

- Obiettivo dell'esercizio 3.1: scrivere un programma per l'analisi sintattica per **espressioni aritmetiche** semplici, scritte in notazione infissa.
  - Elementi: numeri non negativi; operatori di somma, sottrazione, moltiplicazione e divisione; simboli di parentesi tonda ( e ).

Cosa vuol dire LL(1)?

## ► Grammatica LL(1)

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

# Analisi sintattica

- Obiettivo dell'esercizio 3.1: scrivere un programma per l'analisi sintattica per **espressioni aritmetiche** semplici, scritte in notazione infissa.
  - Elementi: numeri non negativi; operatori di somma, sottrazione, moltiplicazione e divisione; simboli di parentesi tonda ( e ).

Cosa vuol dire LL(1)?

- Left to right
- Leftmost derivation
- 1 simbolo alla volta

## ► Grammatica LL(1)

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

# Analisi sintattica

- Obiettivo dell'esercizio 3.1: scrivere un programma per l'analisi sintattica per **espressioni aritmetiche** semplici, scritte in notazione infissa.
  - Elementi: numeri non negativi; operatori di somma, sottrazione, moltiplicazione e divisione; simboli di parentesi tonda ( e ).

link con la teoria

## ► Grammatica LL(1)

|    |  |
|----|--|
| E  | $\langle start \rangle ::= \langle expr \rangle EOF$   |
| E' | $\langle expr \rangle ::= \langle term \rangle \langle exprp \rangle$  |
|    | $\begin{array}{l} \langle exprp \rangle ::= + \langle term \rangle \langle exprp \rangle \\ \quad \quad \quad   - \langle term \rangle \langle exprp \rangle \\ \quad \quad \quad   \varepsilon \end{array}$ |
| T  | $\langle term \rangle ::= \langle fact \rangle \langle termp \rangle$  |
| T' | $\langle termp \rangle ::= \begin{array}{l} * \langle fact \rangle \langle termp \rangle \\ \quad \quad \quad   / \langle fact \rangle \langle termp \rangle \\ \quad \quad \quad   \varepsilon \end{array}$ |
|    | $\langle fact \rangle ::= ( \langle expr \rangle ) \mid NUM$   |

# Discesa ricorsiva

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

**Calcolare insieme guida!**

$$\begin{aligned}\langle start \rangle &::= \langle expr \rangle \text{ EOF} \\ \langle expr \rangle &::= \langle term \rangle \langle exprp \rangle \\ \langle exprp \rangle &::= \begin{array}{l} + \langle term \rangle \langle exprp \rangle \\ - \langle term \rangle \langle exprp \rangle \\ \varepsilon \end{array} \\ \langle term \rangle &::= \langle fact \rangle \langle termp \rangle \\ \langle termp \rangle &::= \begin{array}{l} * \langle fact \rangle \langle termp \rangle \\ / \langle fact \rangle \langle termp \rangle \\ \varepsilon \end{array} \\ \langle fact \rangle &::= ( \langle expr \rangle ) \mid \text{ NUM} \end{aligned}$$

```
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 ...
}
```

# Discesa ricorsiva

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

|                         |       |   |
|-------------------------|-------|---|
| $\langle start \rangle$ | $::=$ | $\langle expr \rangle EOF$  |
| $\langle expr \rangle$  | $::=$ | $\langle term \rangle \langle exprp \rangle$  |
| $\langle exprp \rangle$ | $::=$ | $+ \langle term \rangle \langle exprp \rangle$<br>$  - \langle term \rangle \langle exprp \rangle$<br>$  \varepsilon$ |
| $\langle term \rangle$  | $::=$ | $\langle fact \rangle \langle termp \rangle$  |
| $\langle termp \rangle$ | $::=$ | $* \langle fact \rangle \langle termp \rangle$<br>$  / \langle fact \rangle \langle termp \rangle$<br>$  \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 ...  
}
```

Dovete usare il Lexer, ma inizialmente nella sua versione di base (2.1). All'esame, sarà richiesta la versione completa.



# Discesa ricorsiva

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

|                         |       |   |
|-------------------------|-------|---|
| $\langle start \rangle$ | $::=$ | $\langle expr \rangle$ EOF  |
| $\langle expr \rangle$  | $::=$ | $\langle term \rangle \langle exprp \rangle$  |
| $\langle exprp \rangle$ | $::=$ | $+ \langle term \rangle \langle exprp \rangle$<br>$  - \langle term \rangle \langle exprp \rangle$<br>$  \varepsilon$ |
| $\langle term \rangle$  | $::=$ | $\langle fact \rangle \langle termp \rangle$  |
| $\langle termp \rangle$ | $::=$ | $* \langle fact \rangle \langle termp \rangle$<br>$  / \langle fact \rangle \langle termp \rangle$<br>$  \varepsilon$ |
| $\langle fact \rangle$  | $::=$ | $( \langle expr \rangle ) \mid \text{NUM}$  |

```
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 ...  
}
```

# Discesa ricorsiva

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

|                         |       |   |
|-------------------------|-------|---|
| $\langle start \rangle$ | $::=$ | $\langle expr \rangle EOF$  |
| $\langle expr \rangle$  | $::=$ | $\langle term \rangle \langle exprp \rangle$  |
| $\langle exprp \rangle$ | $::=$ | $+ \langle term \rangle \langle exprp \rangle$<br>$  - \langle term \rangle \langle exprp \rangle$<br>$  \varepsilon$ |
| $\langle term \rangle$  | $::=$ | $\langle fact \rangle \langle termp \rangle$  |
| $\langle termp \rangle$ | $::=$ | $* \langle fact \rangle \langle termp \rangle$<br>$  / \langle fact \rangle \langle termp \rangle$<br>$  \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 ...  
}
```

# Discesa ricorsiva

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

$$\langle start \rangle ::= \langle expr \rangle EOF$$
$$\langle expr \rangle ::= \langle term \rangle \langle exprp \rangle$$
$$\begin{array}{l} \langle exprp \rangle ::= + \langle term \rangle \langle exprp \rangle \\ \quad \quad | - \langle term \rangle \langle exprp \rangle \\ \quad \quad | \varepsilon \end{array}$$
$$\langle term \rangle ::= \langle fact \rangle \langle termp \rangle$$
$$\begin{array}{l} \langle termp \rangle ::= * \langle fact \rangle \langle termp \rangle \\ \quad \quad | / \langle fact \rangle \langle termp \rangle \\ \quad \quad | \varepsilon \end{array}$$
$$\langle fact \rangle ::= ( \langle expr \rangle ) \mid NUM$$

```
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 ...  
}
```

# Discesa ricorsiva

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

|  |
|--|
| $\langle start \rangle ::= \langle expr \rangle EOF$   |
| $\langle expr \rangle ::= \langle term \rangle \langle exprp \rangle$  |
| $\langle exprp \rangle ::= \begin{array}{l} + \langle term \rangle \langle exprp \rangle \\ - \langle term \rangle \langle exprp \rangle \\ \varepsilon \end{array}$ |
| $\langle term \rangle ::= \langle fact \rangle \langle termp \rangle$  |
| $\langle termp \rangle ::= \begin{array}{l} * \langle fact \rangle \langle termp \rangle \\ / \langle fact \rangle \langle termp \rangle \\ \varepsilon \end{array}$ |
| $\langle fact \rangle ::= ( \langle expr \rangle ) \mid NUM$   |

```
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 ...  
}
```

# Discesa ricorsiva

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

|                         |       |   |
|-------------------------|-------|---|
| $\langle start \rangle$ | $::=$ | $\langle expr \rangle$ EOF  |
| $\langle expr \rangle$  | $::=$ | $\langle term \rangle \langle exprp \rangle$  |
| $\langle exprp \rangle$ | $::=$ | $+ \langle term \rangle \langle exprp \rangle$<br>$  - \langle term \rangle \langle exprp \rangle$<br>$  \varepsilon$ |
| $\langle term \rangle$  | $::=$ | $\langle fact \rangle \langle termp \rangle$  |
| $\langle termp \rangle$ | $::=$ | $* \langle fact \rangle \langle termp \rangle$<br>$  / \langle fact \rangle \langle termp \rangle$<br>$  \varepsilon$ |
| $\langle fact \rangle$  | $::=$ | $( \langle expr \rangle ) \mid \text{NUM}$  |

```
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 ...  
}
```

# Discesa ricorsiva

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

|                         |       |   |
|-------------------------|-------|---|
| $\langle start \rangle$ | $::=$ | $\langle expr \rangle EOF$  |
| $\langle expr \rangle$  | $::=$ | $\langle term \rangle \langle exprp \rangle$  |
| $\langle exprp \rangle$ | $::=$ | $+ \langle term \rangle \langle exprp \rangle$<br>$  - \langle term \rangle \langle exprp \rangle$<br>$  \varepsilon$ |
| $\langle term \rangle$  | $::=$ | $\langle fact \rangle \langle termp \rangle$  |
| $\langle termp \rangle$ | $::=$ | $* \langle fact \rangle \langle termp \rangle$<br>$  / \langle fact \rangle \langle termp \rangle$<br>$  \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 ...  
}
```

# Discesa ricorsiva

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

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

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

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

# Discesa ricorsiva

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

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

- Codice:

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

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



# Discesa ricorsiva

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

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

- Codice:

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

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

# Discesa ricorsiva

- **Richiamo teoria:** pseudo-codice per la parsificazione a discesa ricorsiva

```
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
var i : int
```

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

```
procedure parse(v : string)
  w ← v$
  i ← 0
  S()
  match($)
```

```
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
```

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

```
procedure parse(v : string)
  w ← v$
  i ← 0
  S()
  match($)
```

```
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

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

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

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

# 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
```

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

# Codice: classe Parser

```
var w : string
var i : int

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

procedure parse(v : string)
  w ← v$
  i ← 0
  S()
  match($)
```

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

// 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 → α<sub>1</sub> | ... | α<sub>n</sub> sono le produzioni per A

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

```
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  $A$ 
```

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

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

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



# Codice: classe Parser

```
procedure A()                                     //  $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$  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()                                     //  $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$  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  $A$ 
```

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

Variable  $\langle start \rangle$  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  $A$ 
```

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

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

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

# Codice: classe Parser

```
procedure A()                                     //  $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$  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  $A$ 
```

|                         |       |   |     |
|-------------------------|-------|---|-----|
| $\langle start \rangle$ | $::=$ | <span style="border: 1px solid black;"><math>\langle expr \rangle</math></span> | 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  $A$ 
```

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

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

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

# Codice: classe Parser

```
procedure A()                                     //  $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$  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  $A$ 
```

$$\begin{array}{lcl} \langle \text{exprp} \rangle & ::= & + \langle \text{term} \rangle \langle \text{exprp} \rangle \\ & & | - \langle \text{term} \rangle \langle \text{exprp} \rangle \\ & & | \varepsilon \end{array}$$

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
```

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

il campo tag corrisponde al  
nome del token (se numerico,  
256 etc.)

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

|                                |       |  |            |
|--------------------------------|-------|--|------------|
| $\langle \text{exprp} \rangle$ | $::=$ | $+ \langle \text{term} \rangle \langle \text{exprp} \rangle$ | $\alpha_1$ |
|                                |       | $- \langle \text{term} \rangle \langle \text{exprp} \rangle$ | $\alpha_2$ |
|                                |       | $\varepsilon$  | $\alpha_3$ |

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

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

# Codice: classe Parser

```
procedure A()                                     //  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$  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  $A$ 
```

|                                |       |  |            |
|--------------------------------|-------|--|------------|
| $\langle \text{exprp} \rangle$ | $::=$ | $+ \langle \text{term} \rangle \langle \text{exprp} \rangle$ | $\alpha_1$ |
|                                |       | $- \langle \text{term} \rangle \langle \text{exprp} \rangle$ | $\alpha_2$ |
|                                |       | $\varepsilon$  | $\alpha_3$ |

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
```

$\alpha_1 \mid \alpha_2 \mid \alpha_3$   
//  $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$  sono le produzioni per  $A$

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

insieme guida  $\rightarrow +$

|                                |       |  |            |
|--------------------------------|-------|--|------------|
| $\langle \text{exprp} \rangle$ | $::=$ | $+ \langle \text{term} \rangle \langle \text{exprp} \rangle$ | $\alpha_1$ |
|                                |       | $- \langle \text{term} \rangle \langle \text{exprp} \rangle$ | $\alpha_2$ |
|                                |       | $\epsilon$   | $\alpha_3$ |

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

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

# Codice: classe Parser

```
procedure  $A()$                                      //  $A \rightarrow \alpha_1 | \alpha_2 | \alpha_3$  sono le produzioni per  $A$   
  if  $w[i] \in \text{GUIDA}(A \rightarrow \alpha_1)$  then  
     $\vdots$   
  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  $X()$   
     $\vdots$   
  else error                                         //  $w[i]$  non è nell'insieme guida di nessuna produzione per  $A$ 
```

importantissimo: segnalare gli errori (e il prima possibile)

# Codice: classe Parser

```
procedure A()                                     //  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$  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  $A$ 
```

|                                |       |  |            |
|--------------------------------|-------|--|------------|
| $\langle \text{exprp} \rangle$ | $::=$ | $+ \langle \text{term} \rangle \langle \text{exprp} \rangle$ | $\alpha_1$ |
|                                |       | $- \langle \text{term} \rangle \langle \text{exprp} \rangle$ | $\alpha_2$ |
|                                |       | $\varepsilon$  | $\alpha_3$ |

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

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

# Messaggi di errore

- DARE INFO UTILI negli errori

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

# Messaggi di errore

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

- Parametro `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 meccanismi per rilevare input erraneo, in modo tale che 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

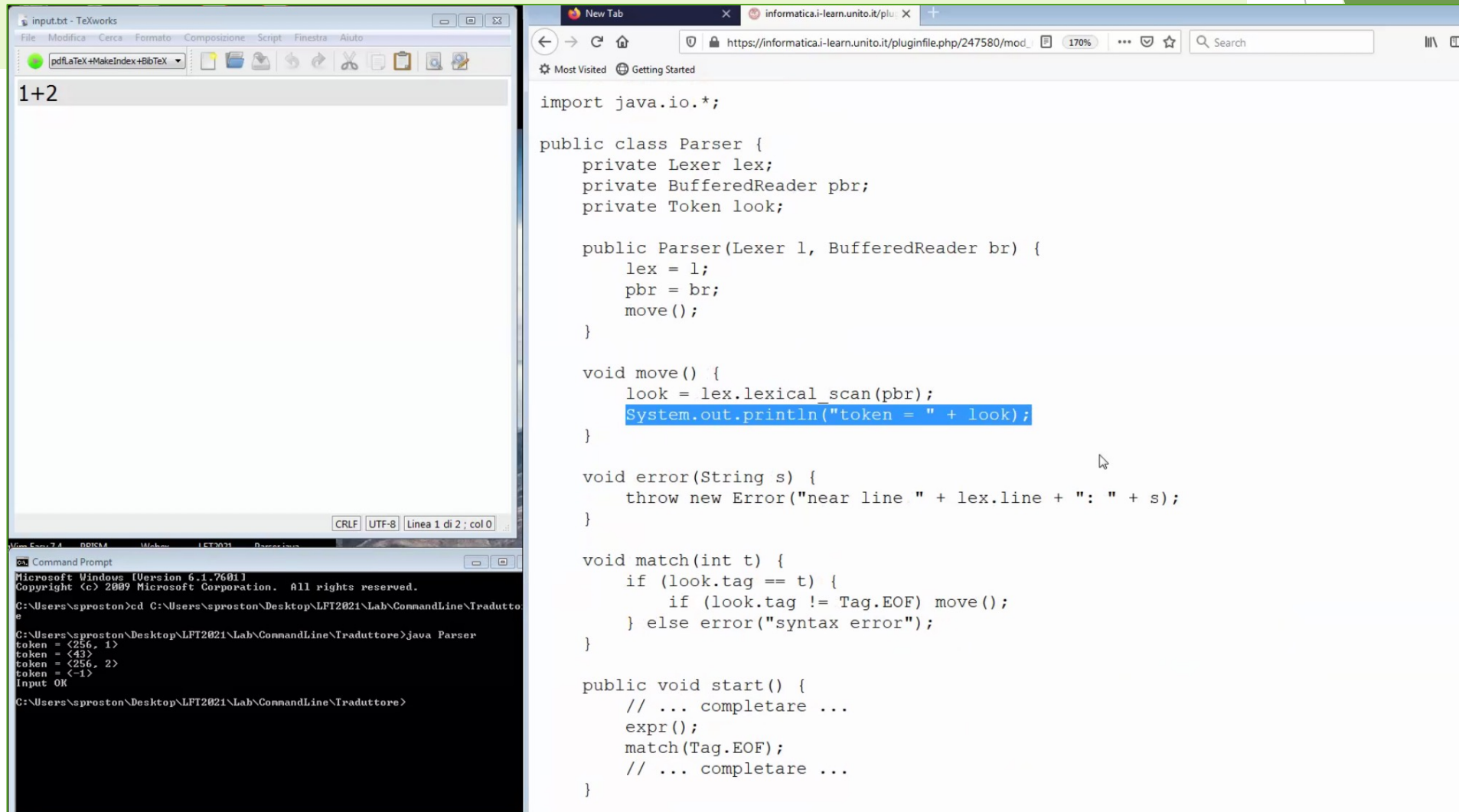
```
procedure parse(v : string)           // v è la stringa da riconoscere
  w ← v$
  i ← 0
  S()                                // S è il simbolo iniziale della grammatica
  match($)                            // controlla di aver letto tutta la stringa
```

- **main**: simile a **parse**; entrambi chiamano la procedura della variabile iniziale della grammatica (**start**).
- Si nota che la fine dell'input è rappresentato esplicitamente con il terminale EOF quindi, nel nostro caso non c'è la necessita di controllare **cc='\$'** 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();}
}
```

**\$ e EOF: dentro start() c'è già il controllo**

# Codice: classe Parser



The image shows a development environment with three windows. The top-left window is a text editor (input.txt - TeXworks) containing the text "1+2". The bottom-left window is a Command Prompt showing the execution of the Java Parser class. The right window is a web browser displaying the source code of the Parser class.

```
import java.io.*;

public class Parser {
    private Lexer lex;
    private BufferedReader pbr;
    private Token look;

    public Parser(Lexer l, BufferedReader br) {
        lex = l;
        pbr = br;
        move();
    }

    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");
    }

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

Command Prompt Output:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\sproston>cd C:\Users\sproston\Desktop\LFT2021\Lab\CommandLine\Traduttore
C:\Users\sproston\Desktop\LFT2021\Lab\CommandLine\Traduttore>java Parser
token = <256, 1>
token = <43>
token = <256, 2>
token = <-1>
Input OK
C:\Users\sproston\Desktop\LFT2021\Lab\CommandLine\Traduttore>
```

Codice: classe Parser

**IMPORTANTE**

**Verificare errori in procedure diverse!**



# Codice: classe Parser

problema nella  
regola di  
produzione **expr**

1+(2\*((9-2))

```
Ca Command Prompt
C:\Users\sproston\Desktop\LFT2021\Lab\CommandLine\Traduttore>java Parser
token = <256, 1>
token = <43>
token = <40>
token = <256, 2>
token = <42>
token = <40>
token = <40>
token = <42>
Exception in thread "main" java.lang.Error: near line 1: Error in grammar <expr>
    at Parser.error(Parser.java:20)
    at Parser.expr(Parser.java:49)
    at Parser.fact(Parser.java:118)
    at Parser.term(Parser.java:80)
    at Parser.expr(Parser.java:45)
    at Parser.fact(Parser.java:118)
    at Parser.term(Parser.java:92)
    at Parser.term(Parser.java:81)
    at Parser.expr(Parser.java:45)
    at Parser.fact(Parser.java:118)
    at Parser.term(Parser.java:80)
    at Parser.expr(Parser.java:57)
    at Parser.expr(Parser.java:46)
    at Parser.start(Parser.java:33)
    at Parser.main(Parser.java:139)
```

# Codice: classe Parser

1\*+2

problema nella  
regola di  
produzione **fact**

```
Exception in thread "main" java.lang.Error: near line 1: Error in grammar <fact>
    at Parser.error(Parser.java:20)
    at Parser.fact(Parser.java:127)
    at Parser.term(Parser.java:92)
    at Parser.term(Parser.java:81)
    at Parser.expr(Parser.java:45)
    at Parser.start(Parser.java:33)
    at Parser.main(Parser.java:139)
C:\Users\sproston\Desktop\LFT2021\Lab\CommandLine\Traduttore>
```

# Parser

```
1 import java.io.*;
2
3 public class Parser {
4     private Lexer lex;
5     private BufferedReader pbr;
6     private Token look;
7
8     public Parser(Lexer l, BufferedReader br) {
9         lex = l;
10        pbr = br;
11        move();
12    }
13
14    void move() {
15        look = lex.lexical_scan(pbr);
16        System.out.println("token = " + look);
17    }
18
19    void error(String s) {
20        throw new Error("near line " + lex.line + ": " + s);
21    }
22
23    void match(int t) {
24        if (look.tag == t) {
25            if (look.tag != Tag.EOF) move();
26        } else error("syntax error");
27    }
28
29    public void start() {
30        // ... completare ...
31        expr();
32        match(Tag.EOF);
33        // ... completare ...
34    }
35
```

```
36 private void expr() {
37     // ... completare ...
38 }
39
40 private void exprp() {
41     switch (look.tag) {
42     case '+':
43         // ... completare ...
44     }
45 }
46
47 private void term() {
48     // ... completare ...
49 }
50
51 private void termp() {
52     // ... completare ...
53 }
54
55 private void fact() {
56     // ... completare ...
57 }
58
59 public static void main(String[] args) {
60     Lexer lex = new Lexer();
61     String path = "...path..."; // il percorso del file da leggere
62     try {
63         BufferedReader br = new BufferedReader(new FileReader(path));
64         Parser parser = new Parser(lex, br);
65         parser.start();
66         System.out.println("Input OK");
67         br.close();
68     } catch (IOException e) {e.printStackTrace();}
69 }
70 }
```