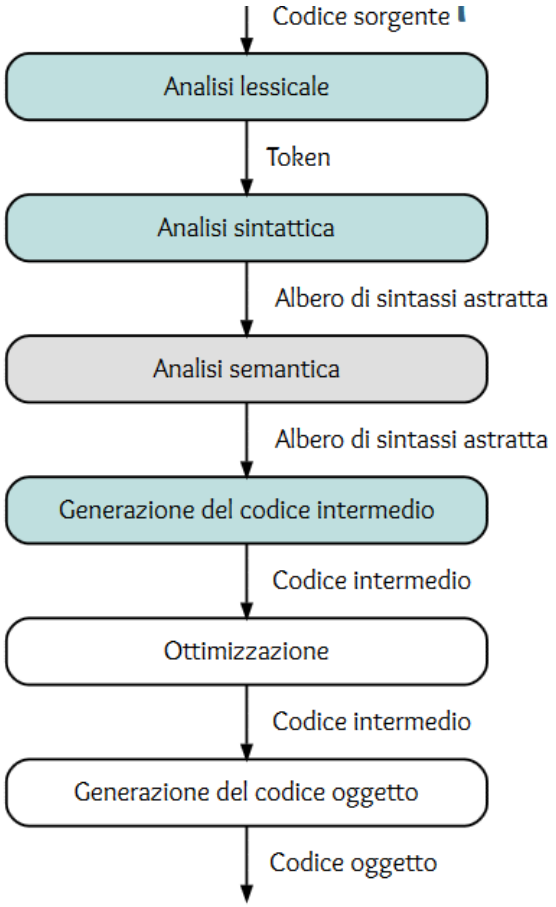
Laboratorio di Linguaggi Formali e Traduttori
LFT lab T2, a.a. 2021/2022

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.

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.

$$
\begin{array}{lcl}
\langle start \rangle & ::= & \langle expr \rangle \ \texttt{EOF} \\
\langle expr \rangle & ::= & \langle term \rangle \ \langle exprp \rangle \\
\langle exprp \rangle & ::= & + \ \langle term \rangle \ \langle exprp \rangle \\
 & | & - \ \langle term \rangle \ \langle exprp \rangle \\
 & | & \varepsilon \\
\langle term \rangle & ::= & \langle fact \rangle \ \langle termp \rangle \\
\langle termp \rangle & ::= & \star \ \langle fact \rangle \ \langle termp \rangle \\
 & | & / \ \langle fact \rangle \ \langle termp \rangle \\
 & | & \varepsilon \\
\langle fact \rangle & ::= & (\ \langle expr \rangle \ ) \ | \ \texttt{NUM}
\end{array}
$$

```
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.

$$
\begin{array}{lcl}
\langle start \rangle & ::= & \langle expr \rangle \ \texttt{EOF} \\
\langle expr \rangle & ::= & \langle term \rangle \ \langle exprp \rangle \\
\langle exprp \rangle & ::= & + \ \langle term \rangle \ \langle exprp \rangle \\
 & | & - \ \langle term \rangle \ \langle exprp \rangle \\
 & | & \varepsilon \\
\langle term \rangle & ::= & \langle fact \rangle \ \langle termp \rangle \\
\langle termp \rangle & ::= & \star \ \langle fact \rangle \ \langle termp \rangle \\
 & | & / \ \langle fact \rangle \ \langle termp \rangle \\
 & | & \varepsilon \\
\langle fact \rangle & ::= & (\ \langle expr \rangle \ ) \ | \ \texttt{NUM}
\end{array}
$$

```
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.

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

```
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.

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

```
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.

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

```
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.

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

```
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.

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

```
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`.

$$\langle start \rangle \quad ::= \quad \langle expr \rangle \ \texttt{EOF}$$

- Codice:
  - Una 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:
  - Una 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`.

$$\langle start \rangle \quad ::= \quad \langle expr \rangle \; \texttt{EOF}$$

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

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

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 ← 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
```

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                          // 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");
}
```

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

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
```

```java
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 → 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
```

$$\langle start \rangle \quad ::= \quad \langle expr \rangle \ \texttt{EOF}$$

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

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

Codice: classe `Parser`

α_k

```
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
```

$$\langle start \rangle \quad ::= \quad \langle expr \rangle \ \texttt{EOF}$$

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

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

Codice: classe `Parser`

```
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
```

α_k

$$\langle start \rangle \quad ::= \quad \langle expr \rangle \ \texttt{EOF}$$

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

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

Codice: classe `Parser`

α_k

```
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
```

$$\langle start \rangle \quad ::= \quad \langle expr \rangle \ \texttt{EOF}$$

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

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

Codice: classe `Parser`

α_k

```
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
```

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

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

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

Codice: classe `Parser`

α_k

```
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
```

$$\langle start \rangle \quad ::= \quad \langle expr \rangle \; \texttt{EOF}$$

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

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

Codice: classe `Parser`

```
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
```

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

Variabile $\langle exprp \rangle$ ha più produzioni

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

Codice: classe `Parser`

$\alpha_1 \mid \alpha_2 \mid \alpha_3$

```
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
```

$\langle exprp \rangle$	::=	$+ \langle term \rangle \langle exprp \rangle$	α_1
	|	$- \langle term \rangle \langle exprp \rangle$	α_2
	|	ε	α_3

Variabile $\langle exprp \rangle$ ha più produzioni

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

Codice: classe `Parser`

$\alpha_1 \mid \alpha_2 \mid \alpha_3$

```
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
```

$\langle exprp \rangle$	::=	$+ \langle term \rangle \langle exprp \rangle$	α_1
	|	$- \langle term \rangle \langle exprp \rangle$	α_2
	|	ε	α_3

Variabile $\langle exprp \rangle$ ha più produzioni

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

Codice: classe `Parser`

$\alpha_1 \mid \alpha_2 \mid \alpha_3$

```
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
```

$\langle exprp \rangle$	::=	$+ \langle term \rangle \langle exprp \rangle$	α_1
	|	$- \langle term \rangle \langle exprp \rangle$	α_2
	|	ε	α_3

Variabile $\langle exprp \rangle$ 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);
}
```

- 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 erroneo, 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; 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();}
}
```