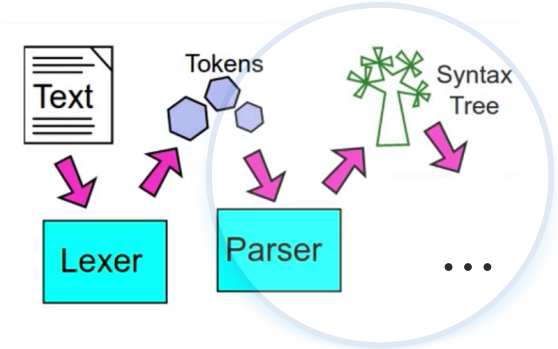
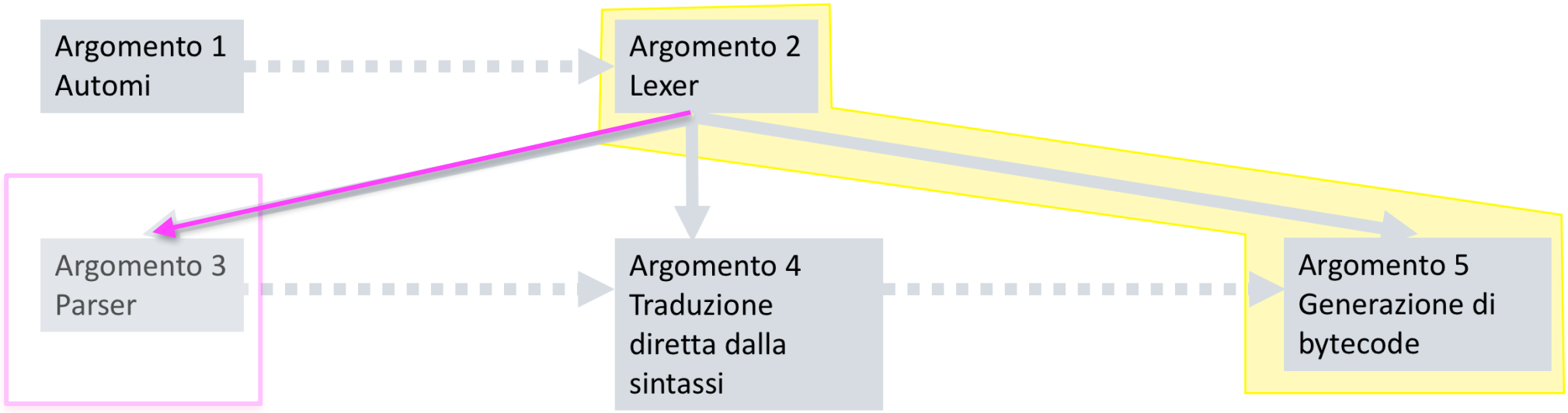


3. Analisi sintattica

Implementazione in Java di un
analizzatore sintattico a discesa ricorsiva
per espressioni di un linguaggio di
programmazione semplice

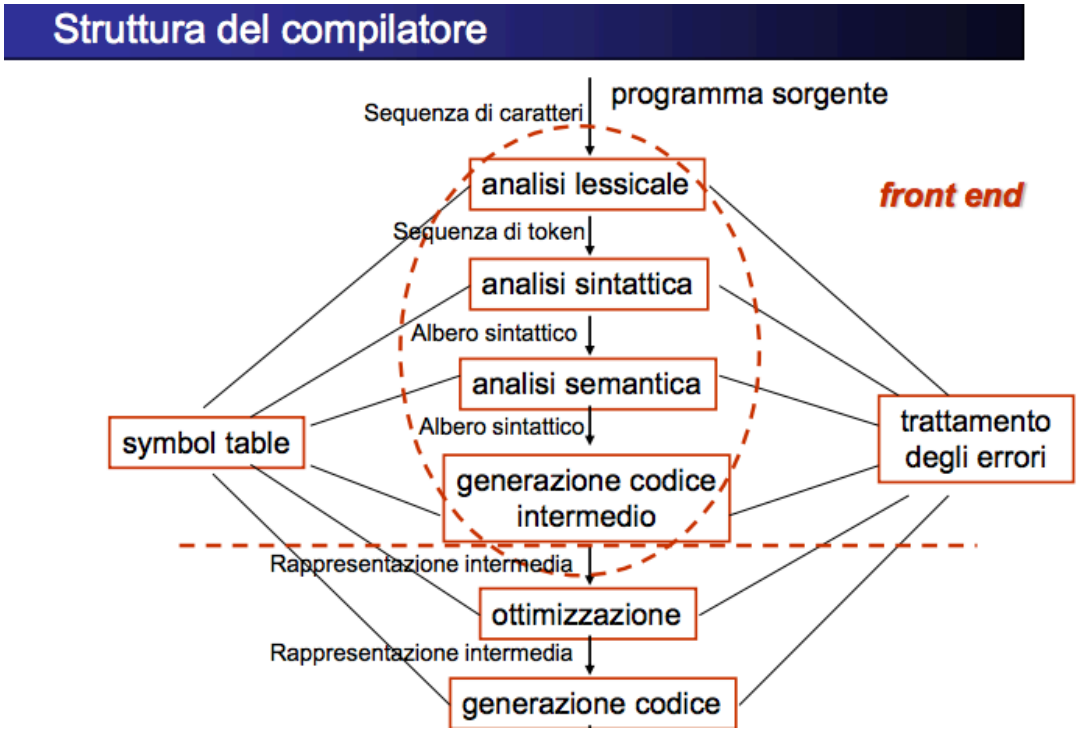
Progetto di laboratorio LFT LAB

- Il progetto di laboratorio consiste in una serie di esercitazioni assistite mirate allo sviluppo di un semplice traduttore.
- Dove siamo:
  - Secondo step: analisi sintattica / parser



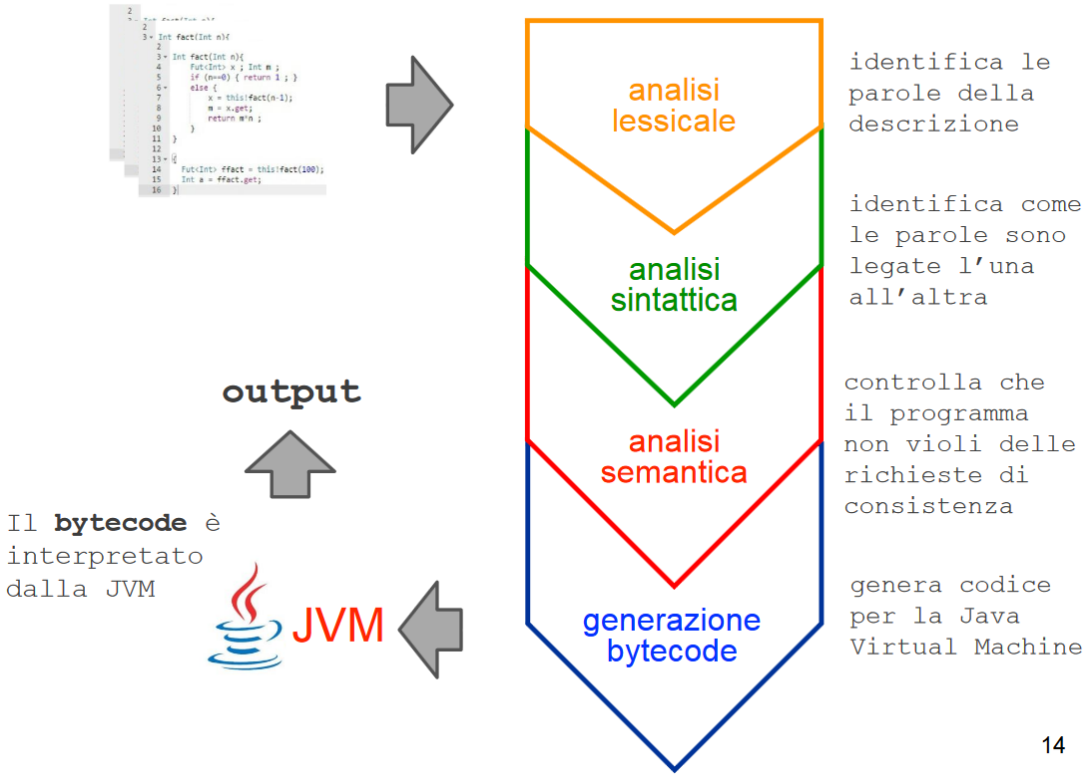
Analizzatore sintattico

- Dove si posiziona in una pipeline di traduzione?
- L'analizzatore lessicale implementato fornirà l'input al programma di analisi sintattica e di traduzione (step successivi all'analisi sintattica).



(...continua...)

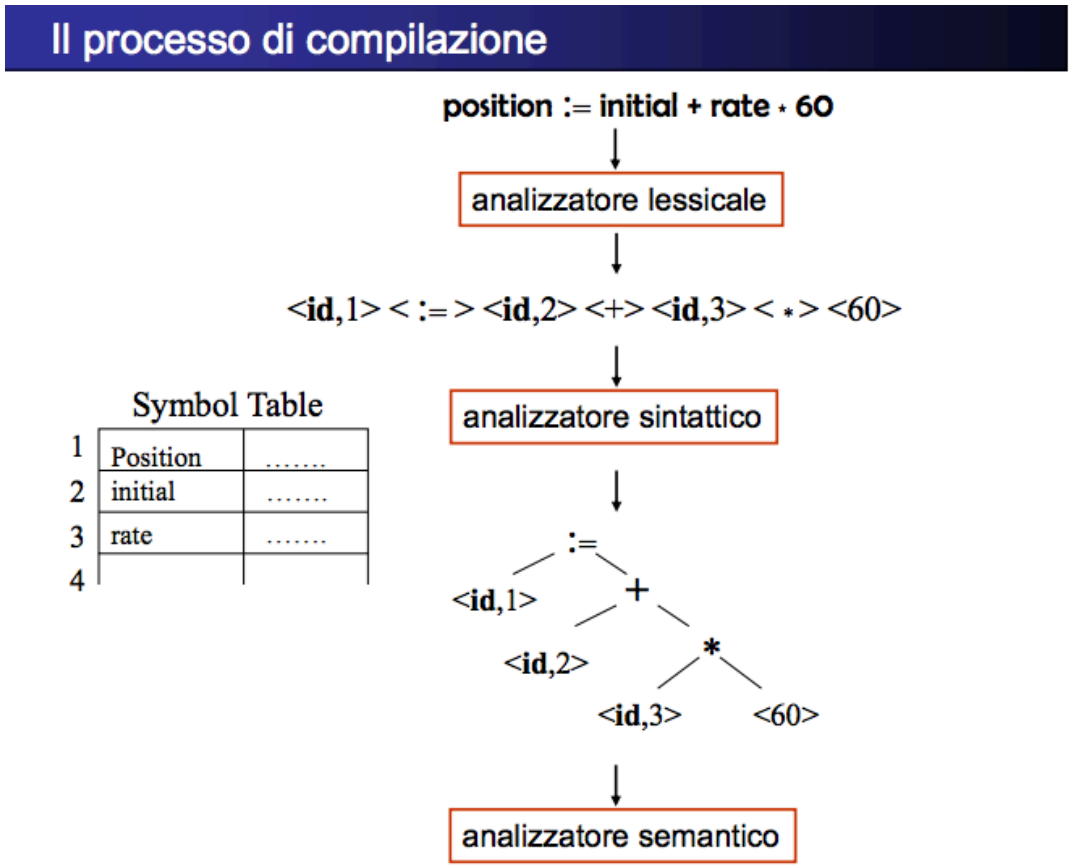
Analizzatore sintattico



- 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.

Analizzatore sintattico

- Dove si posiziona in una pipeline di traduzione?
- L'analizzatore lessicale implementato fornirà l'input al programma di analisi sintattica e di traduzione (step successivi all'analisi sintattica).



Esercizio 3.1

- Si scriva un analizzatore sintattico a discesa ricorsiva che parsifichi espressioni aritmetiche molto semplici, scritte in notazione infissa, composte soltanto da
  - numeri non negativi (ovvero sequenze di cifre decimali)
  - operatori di somma e sottrazione: + –
  - operatori di moltiplicazione e divisione: * /
  - simboli di parentesi: ( )

Esercizio 3.1

- In particolare, l'analizzatore deve riconoscere le espressioni generate dalla grammatica:

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

$$\text{E} \leftarrow \langle expr \rangle ::= \langle term \rangle \langle exprp \rangle$$

$$\text{E'} \leftarrow \langle exprp \rangle ::= + \langle term \rangle \langle exprp \rangle \mid - \langle term \rangle \langle exprp \rangle \mid \varepsilon$$

$$\text{T} \leftarrow \langle term \rangle ::= \langle fact \rangle \langle termp \rangle$$

$$\text{T'} \leftarrow \langle termp \rangle ::= \star \langle fact \rangle \langle termp \rangle \mid / \langle fact \rangle \langle termp \rangle \mid \varepsilon$$

$$\text{F} \leftarrow \langle fact \rangle ::= (\langle expr \rangle) \mid \texttt{NUM}$$

Grammatica (LL(1)):

Esercizio 3.1

- Il programma deve fare uso dell'analizzatore lessicale sviluppato in precedenza.
- Si noti che l'insieme di token corrispondente alla grammatica di questa sezione è un sottoinsieme dell'insieme di token corrispondente alle regole lessicali specificate nella Sezione 2.

Q. *Quale analizzatore lessicale utilizziamo?*
Abbiamo diversi esercizi nella sezione 2 sull'analisi lessicale.
A. Gli esercizi della sezione 2 sono intesi in modo incrementale.
Per cominciare, potete usare il primo lexer sviluppato (2.1).
Quando venite a discutere il progetto d'esame mi aspetto che sia agganciato il lexer completo (che riconosce identificatori con pattern definito in 2.2. e riconosce e ignora commenti come specificato in 2.3)

Esercizio 3.1

- Nei casi in cui l'input ***non*** corrisponda alla grammatica, l'output del programma deve consistere di un messaggio di errore (come illustrato nelle lezioni di teoria) indicando la procedura in esecuzione quando l'errore è stato individuato.

Richiamo teoria

Grammatiche LL(1) e parsificazione deterministica look ahead

- Pseudo-codice per la parsificazione a discesa ricorsiva

Ad *ogni variabile* A con produzioni
$A \rightarrow \alpha_1$
$A \rightarrow \alpha_2$
...
$A \rightarrow \alpha_k$
si associa una procedura:

```
A()
  begin
        if (cc ∈ Gui (A → α1)) body (α1)
        else if (cc ∈ Gui(A → α2)) body (α2)
         ....
        else if (cc ∈ Gui(A → αk)) body (αk)
        else ERRORE (…)
  end
```

Se $\alpha = \varepsilon$, body(ε) = do nothing

Se $\alpha = X_1, \ldots, X_m$, body($X_1, \ldots, X_m$) è così definito:

body($X_1, \ldots, X_m$) = act (X_1) act (X_2) … act (X_m)

$$act(X) = \begin{cases} X() & \text{se } X \in V \\ \text{if (cc = X) cc} \leftarrow \text{PROSS} & \text{se } X \in \Sigma \\ \text{else ERRORE(…)} & \end{cases}$$

Discesa ricorsiva

- Una procedura associata ad ogni variabile
- Procedura di un variabile:
  codice derivato 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 derivato 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
$\langle expr \rangle$	::=	$\langle term \rangle \langle exprp \rangle$
$\langle exprp \rangle$	::=	$+ \langle term \rangle \langle exprp \rangle$
	|	$- \langle term \rangle \langle exprp \rangle$
	|	ε
$\langle term \rangle$	::=	$\langle fact \rangle \langle termp \rangle$
$\langle termp \rangle$	::=	$\star \langle fact \rangle \langle termp \rangle$
	|	$/ \langle fact \rangle \langle termp \rangle$
	|	ε
$\langle fact \rangle$	::=	($\langle expr \rangle$) | 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 derivato 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 derivato 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 derivato 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 derivato 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
$\langle expr \rangle$	::=	$\langle term \rangle \langle exprp \rangle$
$\langle exprp \rangle$	::=	$+ \langle term \rangle \langle exprp \rangle$
	|	$- \langle term \rangle \langle exprp \rangle$
	|	ε
$\langle term \rangle$	::=	$\langle fact \rangle \langle termp \rangle$
$\langle termp \rangle$	::=	$\star \langle fact \rangle \langle termp \rangle$
	|	$/ \langle fact \rangle \langle termp \rangle$
	|	ε
$\langle fact \rangle$	::=	($\langle expr \rangle$) | 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 ...
}
```

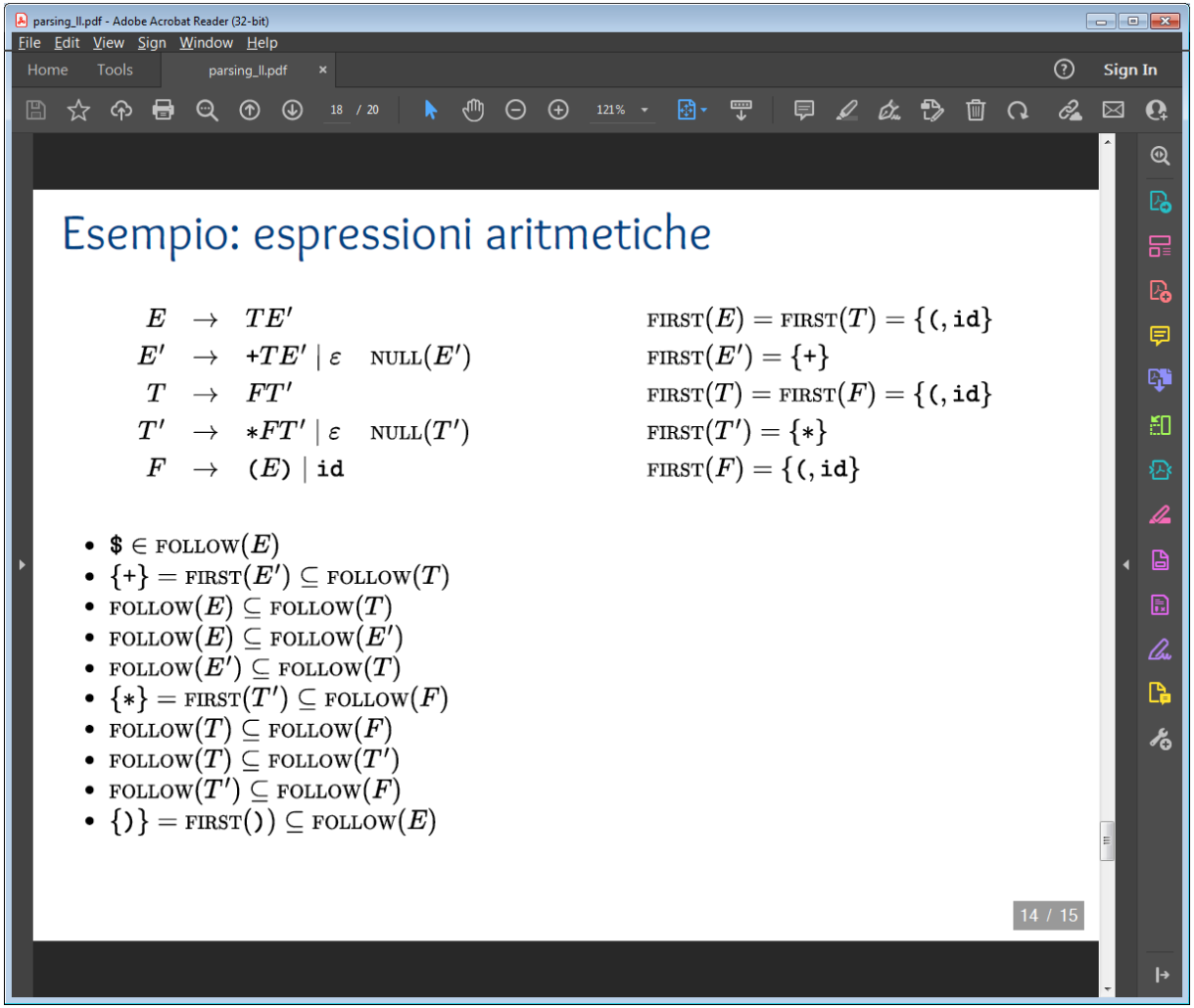
Discesa ricorsiva

- Una procedura associata ad ogni variabile.
- Procedura di un variabile:
  codice derivato 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 ...
}
```

Richiamo teoria



id -> NUM

Codice di partenza

```
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();
```

Usa l'output del lexer:

si muove token per token

stampa i token

circostanziare l'errore

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 gli errori siano 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

Se $\alpha = \varepsilon$, body(ε) = do nothing

Se $\alpha = X_1, \ldots, X_m$, body($X_1, \ldots, X_m$) è così definito:

body($X_1, \ldots, X_m$) = act (X_1) act (X_2) … act (X_m)

act(X) = { X() se $X \in V$; if (cc = X) cc ← PROSS else ERRORE(…) se $X \in \Sigma$

```
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
cc	look
X (se X è un terminale)	t
cc ← PROSS	move()
ERRORE	error()

Codice di partenza

```
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 ...
}
```

- **Suggerimenti su come procedere:**
- calcolare l'insieme guida
- seguire il pattern:

  if then … else errore
- errore circostanziato: indicare

  la produzione in cui si e' verificato l'errore.

  Più informativo!

Codice di partenza

```
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();}
}
```

Codice: classe Parser

- main: chiamare la procedura della variabile iniziale della grammatica (⟨*start*⟩).
- Si noti che la fine dell'input è rappresentata esplicitamente con il terminale EOF quindi, nel nostro contesto, 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();}
}
```

```
main()   //Program discesa_ricorsiva
begin cc ← PROSS
        S()
        if (cc = '$')  "stringa accettata"
        else ERRORE(…)
end
```