

LeX & Yacc per Java

Linguaggi e Computabilità
A.A. 2016-2017

UNA BREVE INTRODUZIONE

Parser & Lexer (1)

- Un **parser** è un analizzatore sintattico;
- È uno dei componenti di base di compilatori e traduttori;
- Analizza un testo per determinare se la sua struttura grammaticale rispetta una certa **grammatica formale**;
 - Verifica la correttezza sintattica
 - Contribuisce all'Identificazione degli errori di sintassi
 - Produce un albero sintattico
 - Il testo è composto da sequenze di *token*

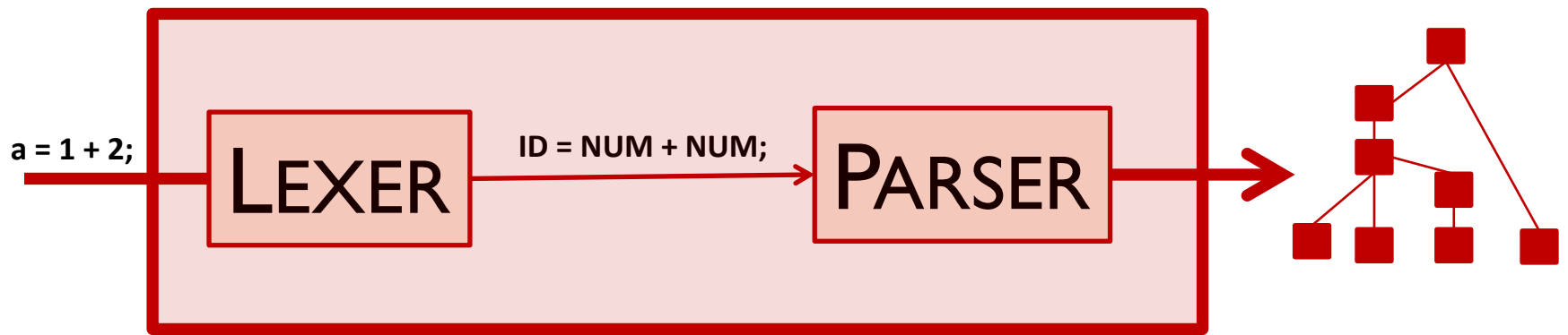
Parser & Lexer (2)

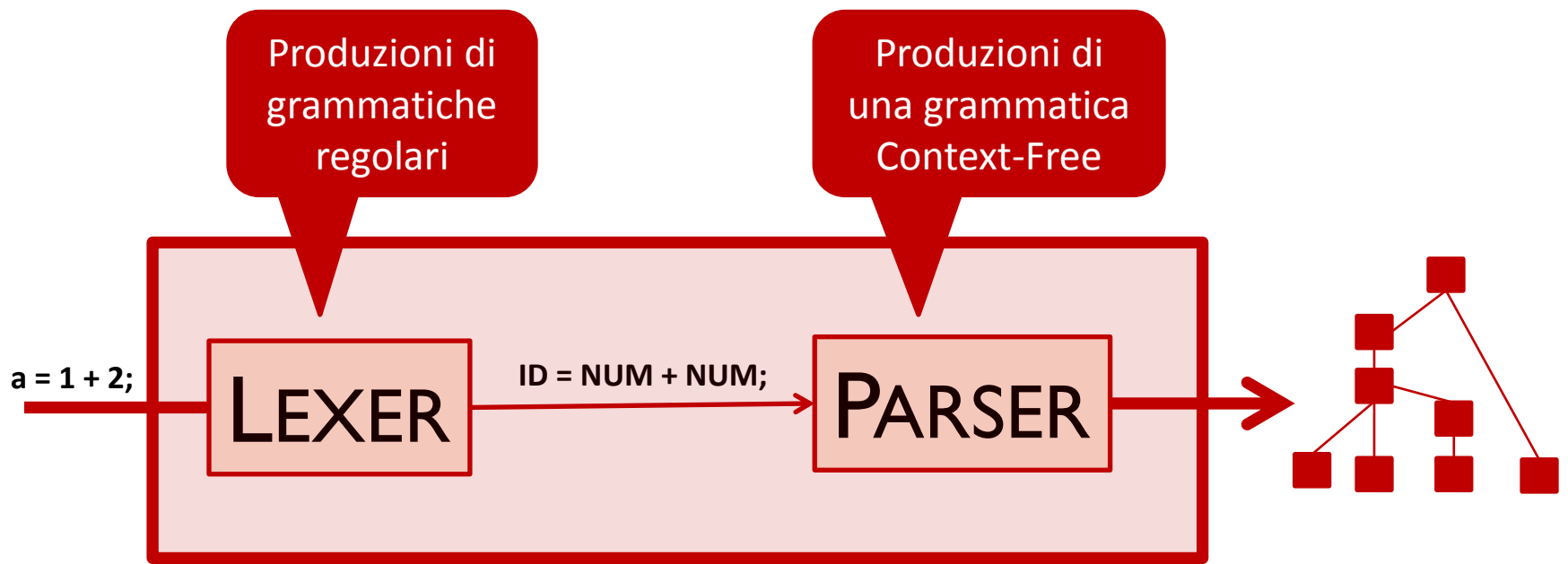
- I parser ricorrono agli **analizzatori lessicali** per identificare le sequenze di *token* a partire da sequenze di caratteri
 - Spesso gli analizzatori lessicali sono tool separati rispetto al parser
 - Gli analizzatori lessicali vengono chiamati **lexer**¹

1. I **lexer** vengono anche chiamati *scanner*

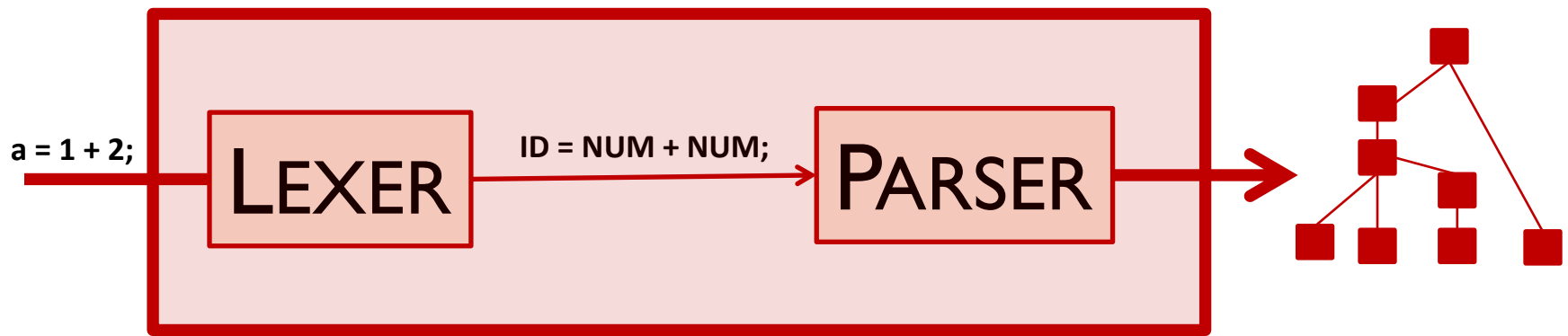
Parser & Lexer (3)

- Possono essere creati manualmente, in gergo «from scratch», come qualsiasi altro software;
- Più spesso vengono creati attraverso tool semi-automatici
 - **generatori di parser:** es., Yacc
 - **generatori di lexer:** es, LeX





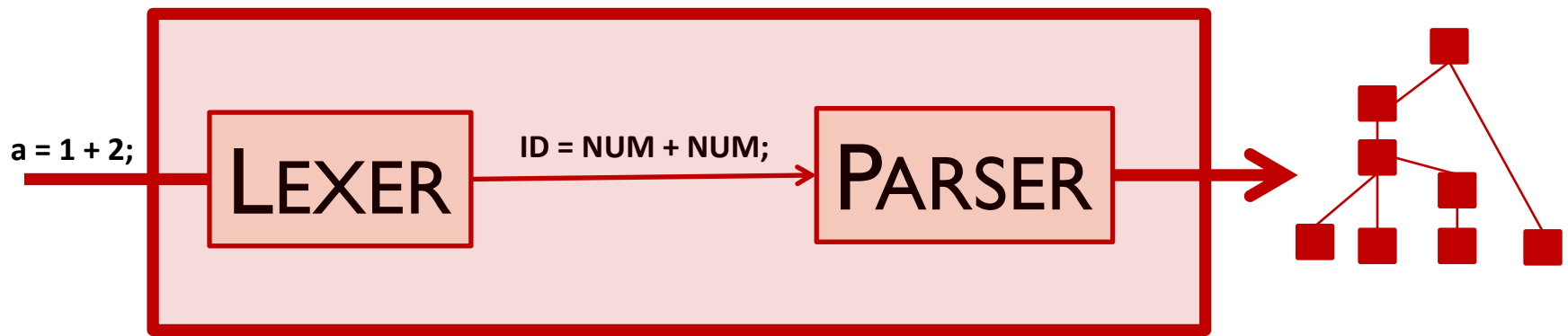
GENERATORE DI PARSER



Sono chiamati anche *compiler-compiler* o *compiler generator*, poiché idealmente producono compilatori; più spesso generano soltanto un componente del compilatore, cioè il *parser*.

Es. YACC, BYACC/J, PLY, Bison, Javacc

GENERATORE DI PARSER

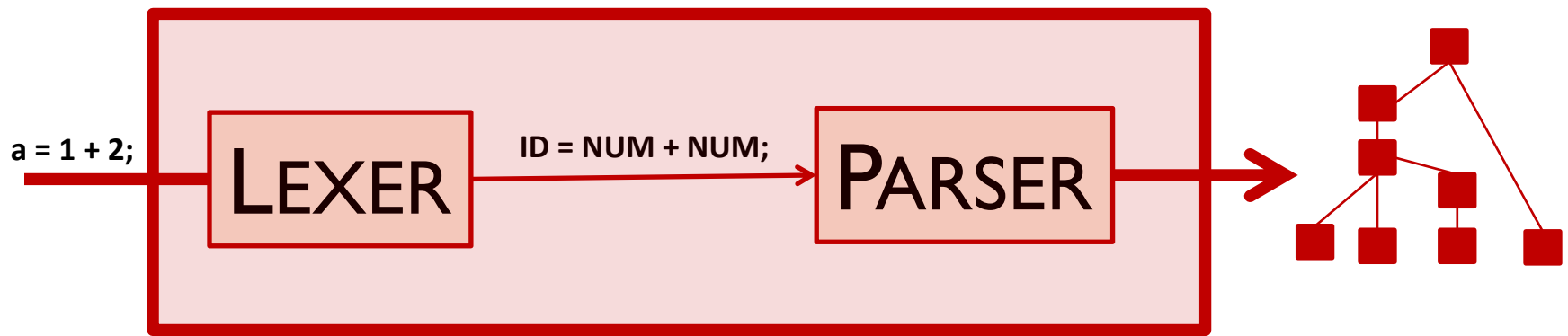


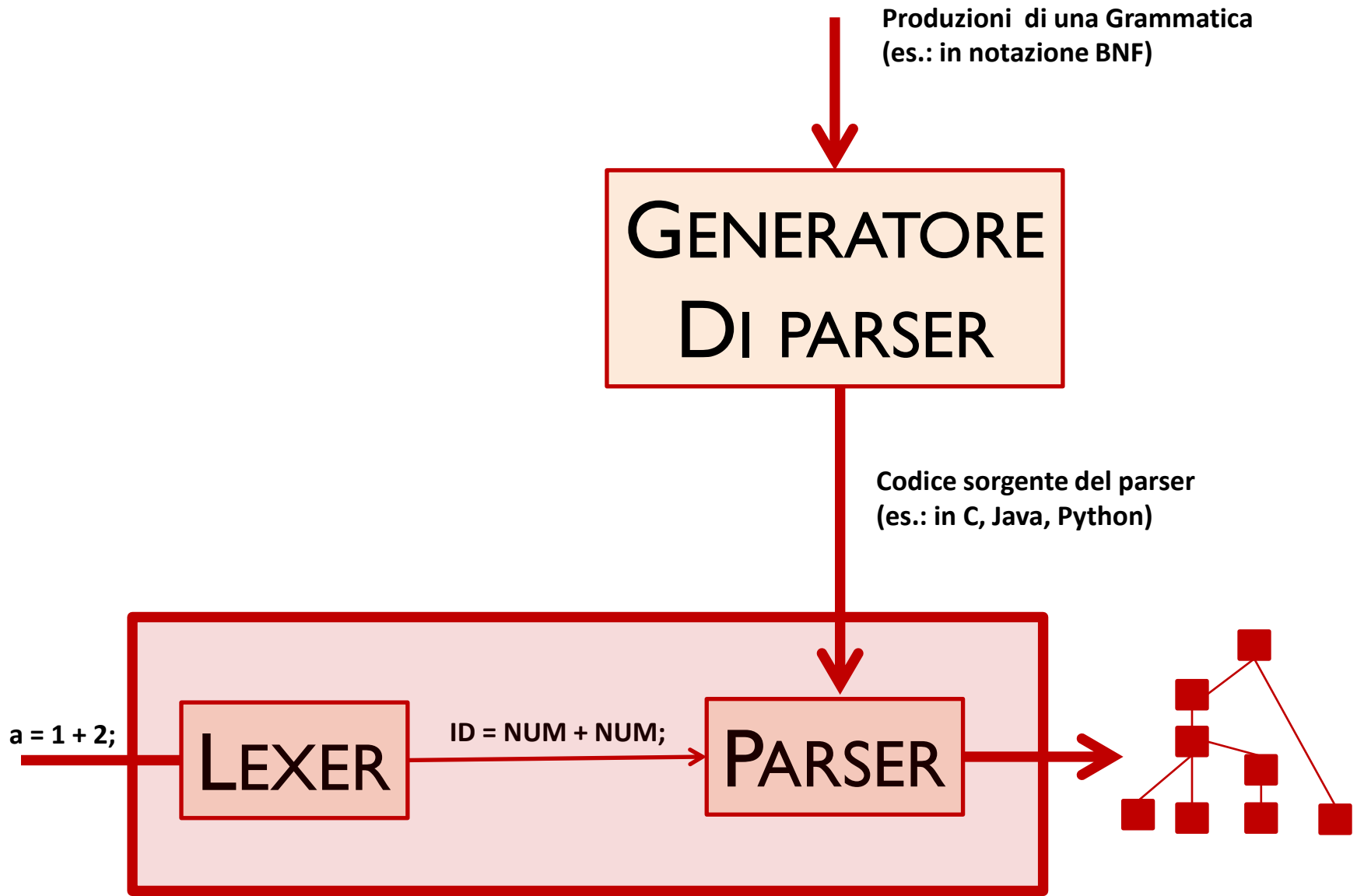
Sono chiamati anche *compiler-compiler* o *compiler generator*, poiché idealmente producono compilatori; più spesso generano soltanto un componente del compilatore, cioè il *parser*.

Es. YACC, BYACC/J, PLY, Bison, Javacc

Genera un programma che implementa un automa a pila.

GENERATORE DI PARSER

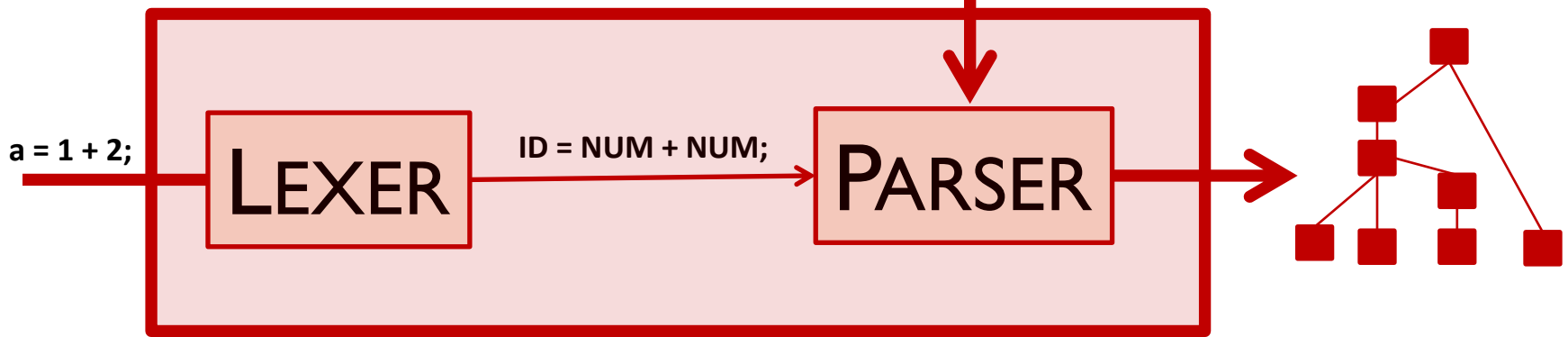




La specifica di un Parser Generator (il suo input) contiene sempre 3 sezioni: dichiarazioni, regole di traduzione, procedure ausiliarie.

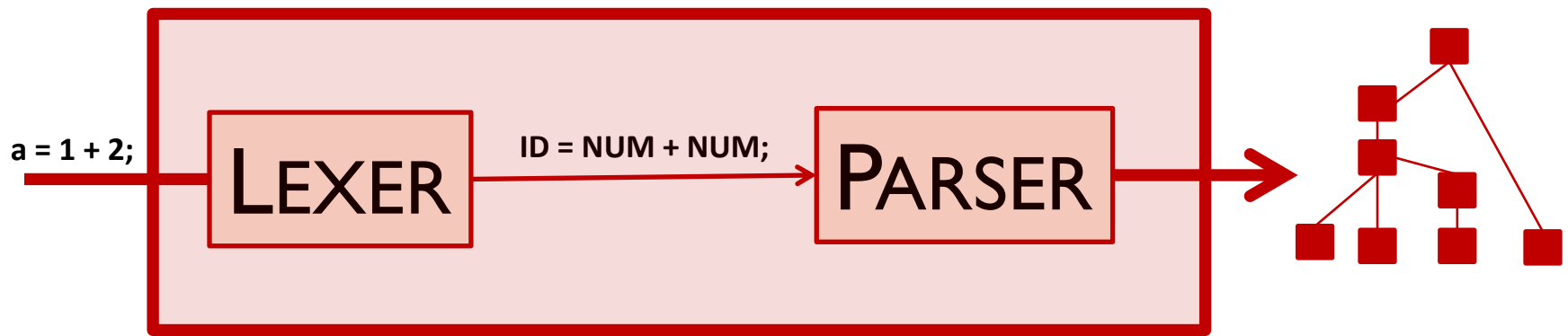


GENERATORE DI PARSER



GENERATORE
DI LEXER

GENERATORE
DI PARSER

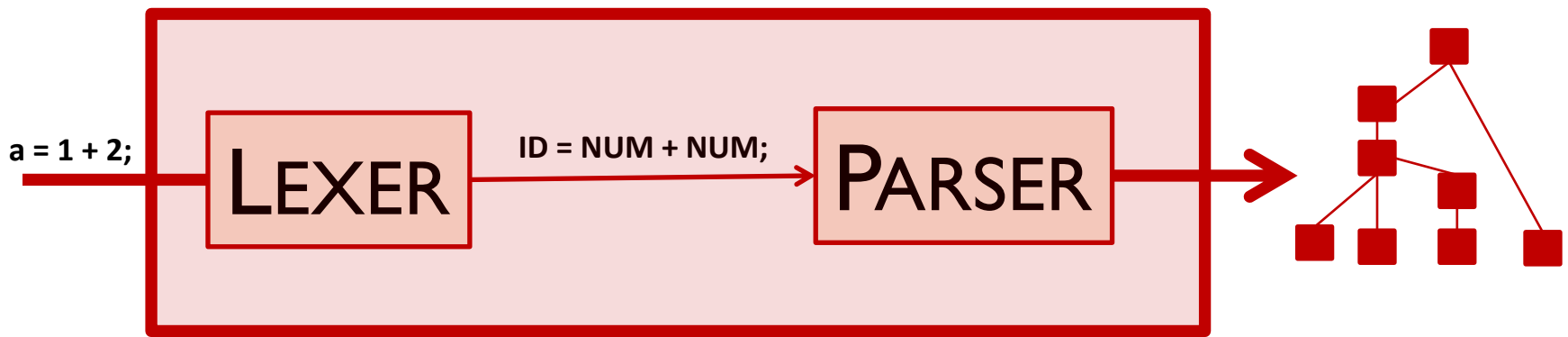


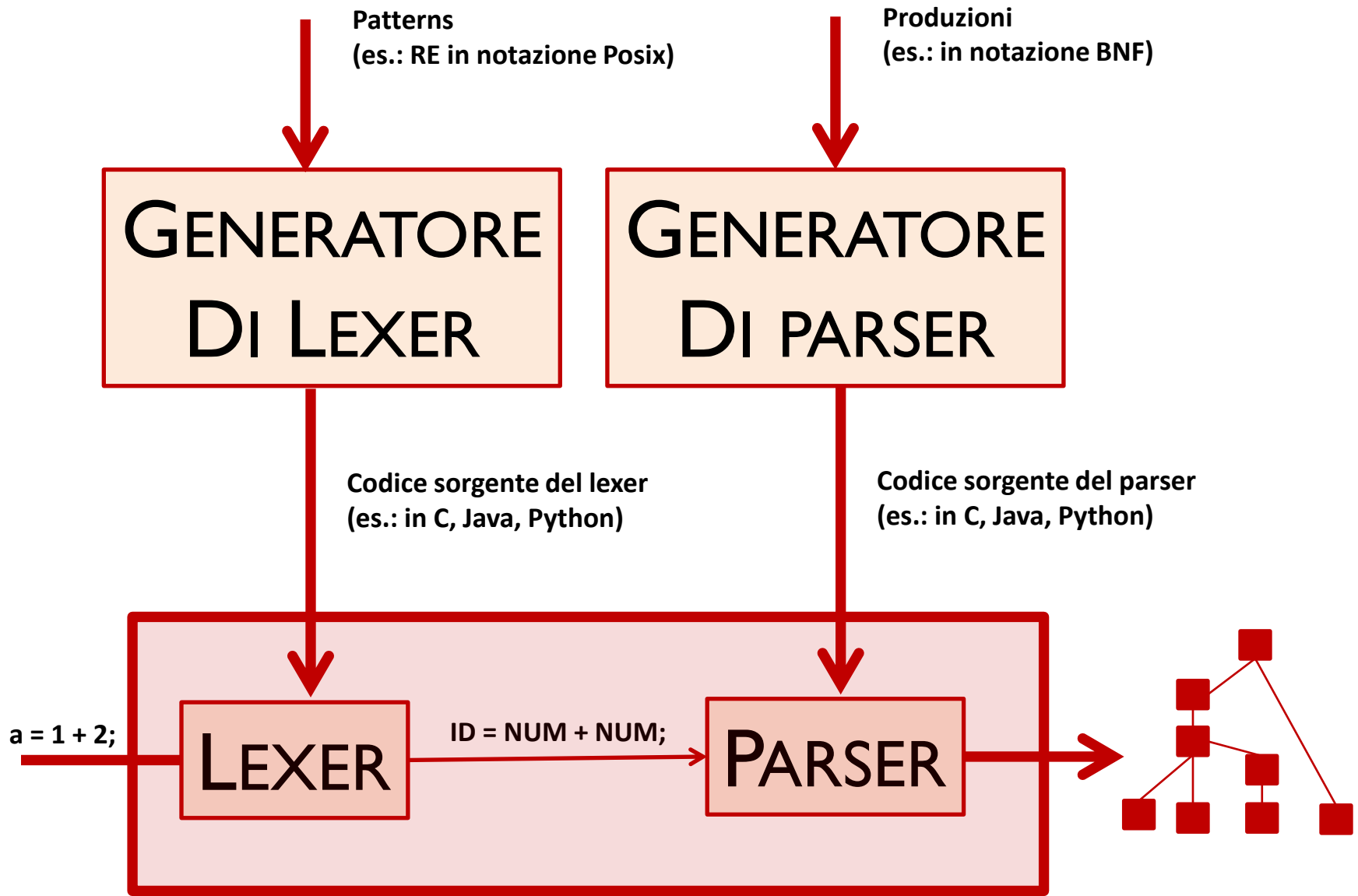
GENERATORE DI LEXER

Questo è un generatore di
analizzatore lessicale (cioè di
lexer o scanner) o «lex
compiler».

Ad es.: Lex

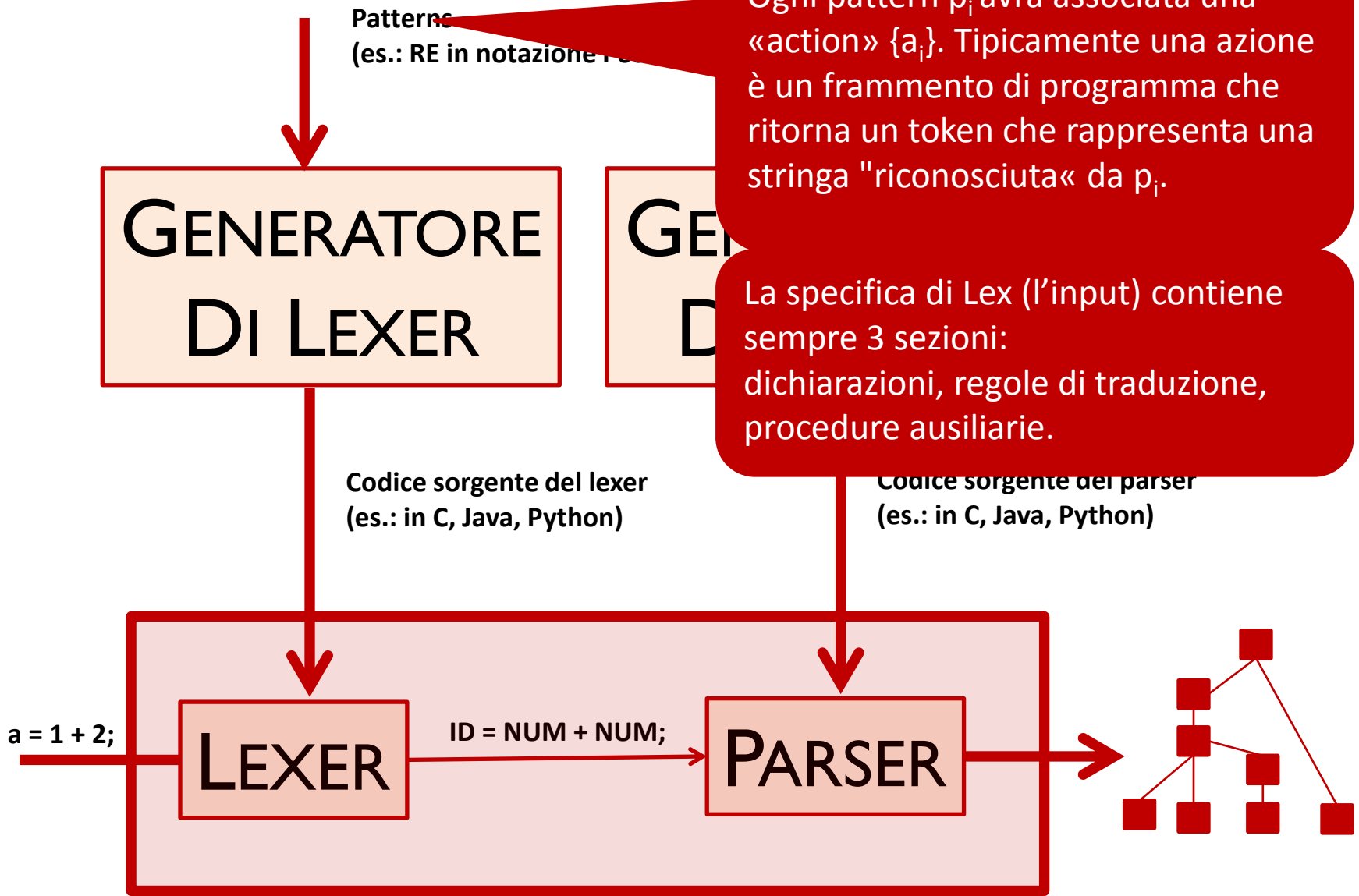
DI PARSER

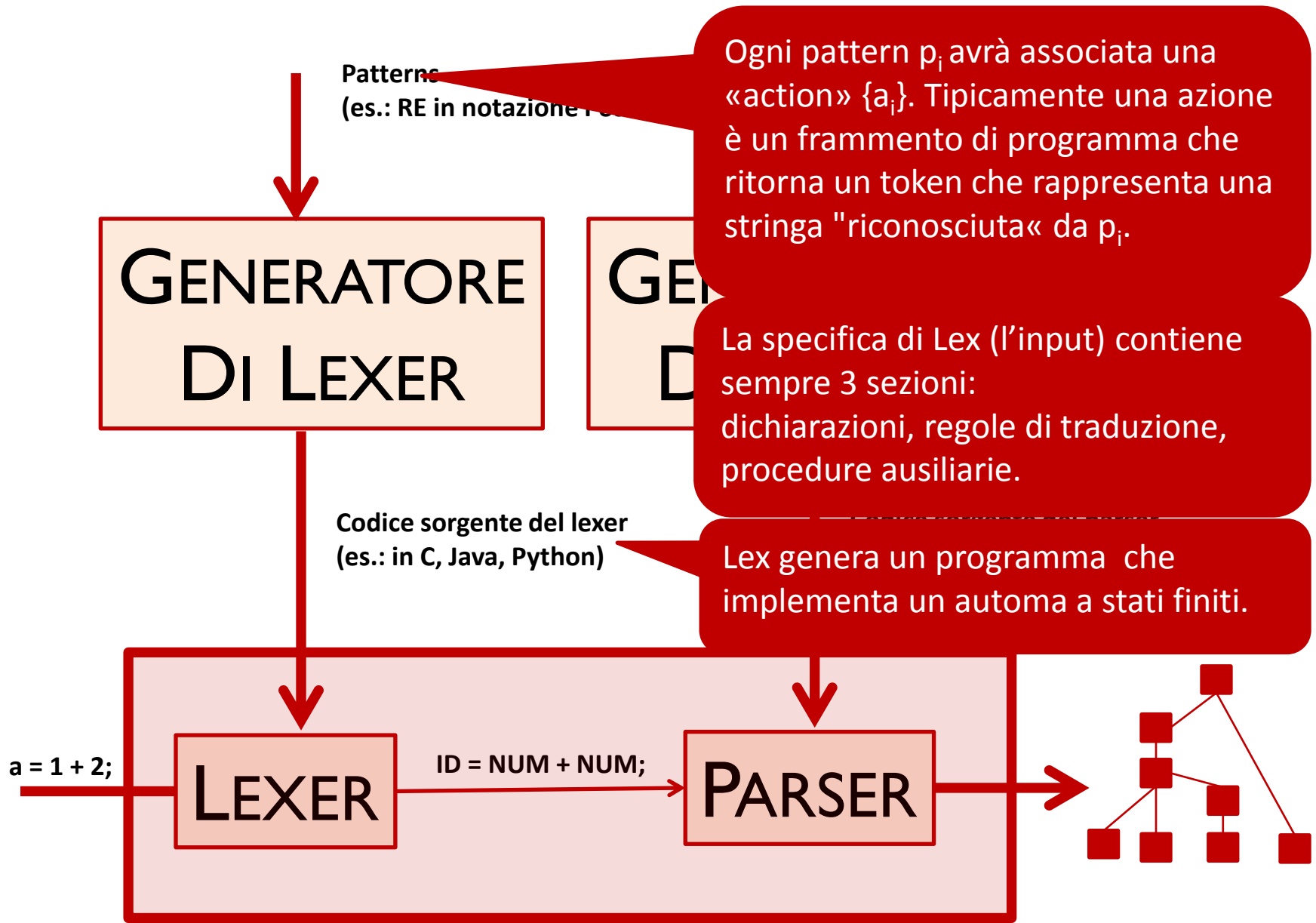




Ogni pattern p_i avrà associata una «action» $\{a_i\}$. Tipicamente una azione è un frammento di programma che ritorna un token che rappresenta una stringa "riconosciuta" da p_i .

La specifica di Lex (l'input) contiene sempre 3 sezioni: dichiarazioni, regole di traduzione, procedure ausiliarie.





Ogni pattern p_i avrà associata una «action» $\{a_i\}$. Tipicamente una azione è un frammento di programma che ritorna un token che rappresenta una stringa "riconosciuta" da p_i .

La specifica di Lex (l'input) contiene sempre 3 sezioni: dichiarazioni, regole di traduzione, procedure ausiliarie.

Lex genera un programma che implementa un automa a stati finiti.

Questo programma è il lexer, chiamato dal parser un token alla volta. Esso analizza l'input (disponibile), un carattere alla volta, finché non trova il prefisso più lungo mecciato da p_i , quindi esegue a_i finché non torna il controllo al parser (insieme al token identificato)

Patterns
(es.: RE in notazione \backslash)

GENERATORE
DI LEXER

GEN
D

Codice sorgente del lexer
(es.: in C, Java, Python)

`a = 1 + 2;`

LEXER

`ID = NUM + NUM;`

LeX & Yacc – Ieri e Oggi

- LeX e Yacc sono nati in ambiente UNIX e producono codice sorgente in linguaggio C
 - Versioni migliorate/attuali
 - LeX ☐ Flex (Fast Lexical Analyzer)
 - Yacc ☐ Bison
 - Flex e Bison **non** sono tool per la sola piattaforma UNIX/Linux
 - Esistono *port* anche per Windows e Macintosh
 - Producono (ancora) codice in linguaggio C/C++
- E gli altri linguaggi di programmazione?
 - Esistono implementazioni di LeX e Yacc per molti linguaggi (es., Java, Python, ecc.)

YACC E JAVA

Yacc e Java

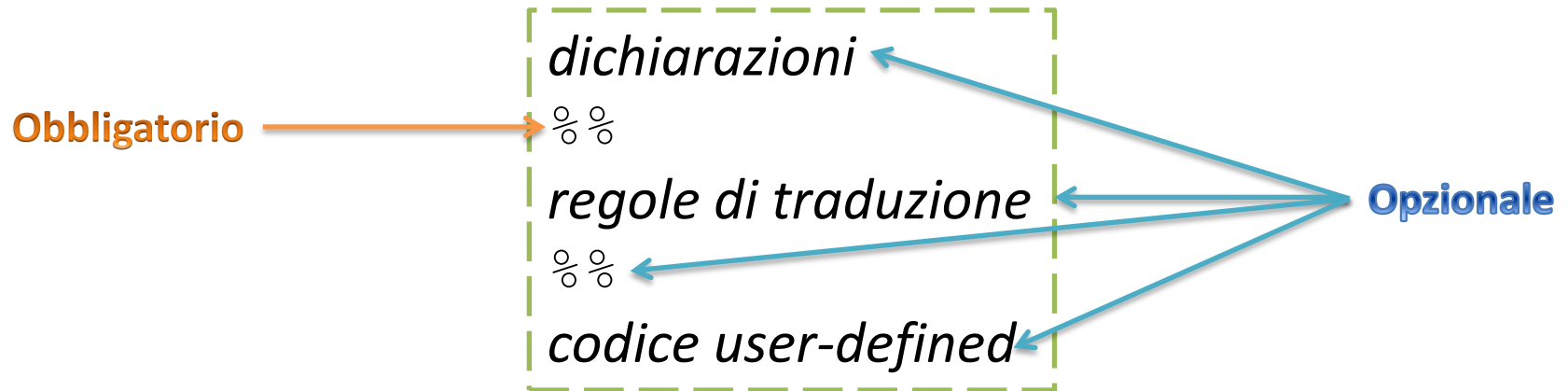
- BYACC/J - Estende il *Berkeley YACC-compatible parser generator* v.1.8
 - Usa una sintassi compatibile con quella di Yacc
 - Può generare (anche) codice sorgente in linguaggio Java (normalmente produce codice in linguaggio C++)
- Per eseguire BYACC/J dalla riga di comando e produrre codice sorgente in linguaggio Java¹

```
yacc -J <input_file>
```

1. Da eseguire dopo essersi posizionati nella cartella `bin` all'interno della cartella di installazione di BYACC/J.

Struttura dei File `.y` in BYACC/J

- Stessa struttura dei file `.y` usati con le versioni di Yacc per generare codice C/C++



- Minimo contenuto valido

`%%`

Cosa fa? Copia tutto il linguaggio preso in input dal *parser* sull'output...

Definizione della Grammatica

- Simboli *non terminali*
 - Convenzione: scritti in **minuscolo**
 - Es.: `exp`, `stmt`, ...
- Simboli *terminali (token)*
 - Convenzione: scritti in **maiuscolo**
 - Es.: `INTEGER`, `FLOAT`, `IF`, `WHILE`, `';`', `'.'`, `'+'`, ecc.
- Regole sintattiche
 - Chiamate ***produzioni***
 - Es.: `exp: exp '+' exp | exp '*' exp;`

Definizione dei Simboli

- Si definiscono nella sezione delle dichiarazioni
 - Non terminali: `%type nome` □ Es.: `%type string_ass`
 - Terminali: `%token NOME` □ Es.: `%token NUM`
- Per *default* tutti i simboli sono di tipo `int`, ma è possibile utilizzare altri tipi di dato
 1. La classe `ParserVal` fornisce dei metodi che permettono l'uso di tipi di dato differenti da `int`
 2. Associando il tipo ai simboli terminali e non terminali attraverso l'uso di parentesi angolari nei costrutti che ne permettono la dichiarazione

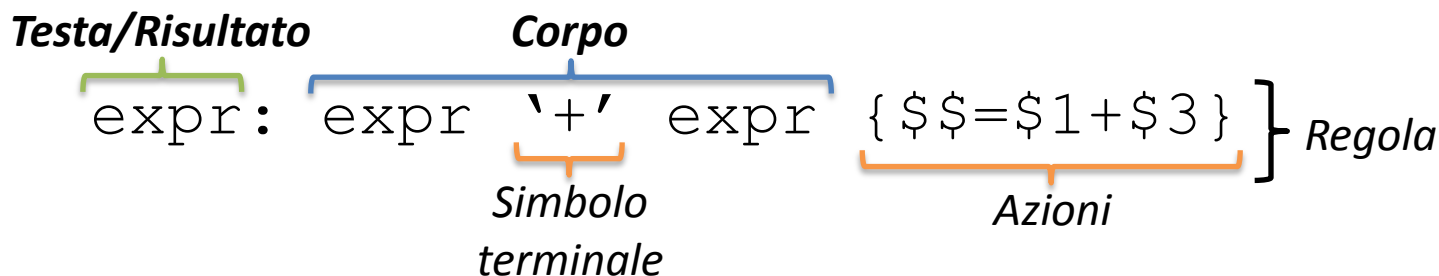
```
%token<dval> NUM DOUBLE
%token<ival> NUM_INT
%type<sval> stringa
%type<oval> oggetto
```

Definizione delle Regole (1)

- Forma:

testa: corpo azioni

- **Testa***: simbolo ***non terminale*** a cui si riduce l'espressione nella *parte destra* della regola
- **Corpo**: è composto da simboli ***terminali, non terminali***
- **Azioni**: insieme di istruzioni Java che vengono eseguite se il corpo della produzione è verificato



* Detta anche ***Risultato***

Definizione delle Regole (2)

- È possibile elencare «parti destre» alternative, che portano alla riscrittura dello stesso simbolo *non terminale* nella parte sinistra

```
expr: expr '+' expr { $$=$1+$3 } |  
      expr '*' expr { $$=$1*$3 } ;
```

- Se la parte destra è vuota, la produzione viene soddisfatta anche dalla stringa vuota

```
expr: /* stringa vuota */ |  
      expr_1;
```

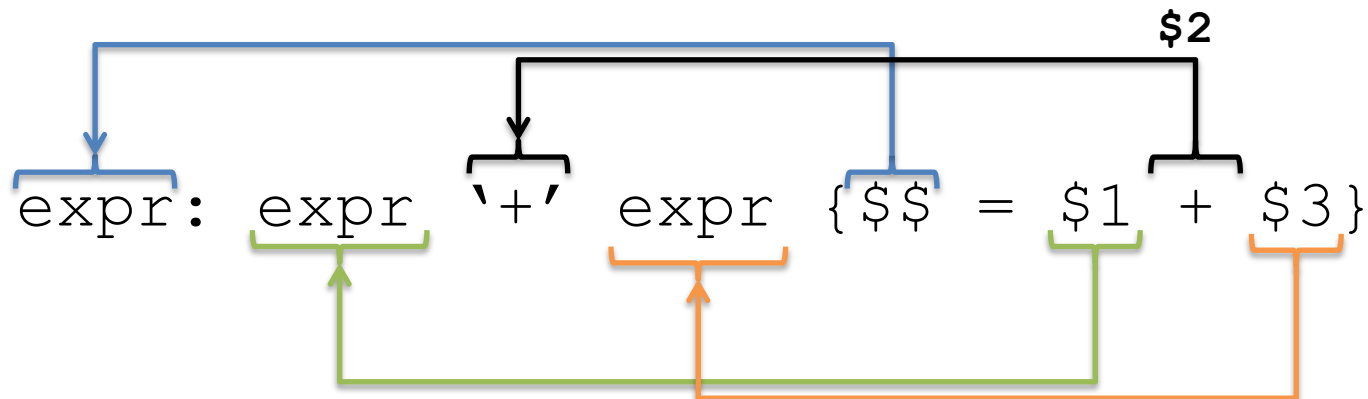
- La produzione è *ricorsiva* se il simbolo non terminale della parte sinistra compare anche nella parte destra

```
expr_seq: expr |  
          expr_seq ',' expr;
```

(Meglio evitare un uso eccessivo della ricorsione, se possibile)

Produzioni e Azioni

- Un'azione è una parte di codice Java che viene eseguita quando la produzione viene applicata (riduzione)



- Il valore dell' n -esimo elemento della produzione corrisponde a $\$n$; $$$$ rappresenta la parte sinistra; se non si specifica nessuna azione, per *default* si ha $$$=S1$
- Il tipo associato a $\$n$ è quello dell'elemento corrispondente
 - Si può forzare un altro tipo usando la sintassi $\$<tipo>n$

Associatività e Precedenza

- È possibile definire l'associatività dei simboli (a sinistra/a destra)

`%left OP`

} $x \text{ OP } y \text{ OP } z \Rightarrow (x \text{ OP } y) \text{ OP } z$

`%right OP`

} $x \text{ OP } y \text{ OP } z \Rightarrow x \text{ OP } (y \text{ OP } z)$

`%nonassoc OP`

} $x \text{ OP } y \text{ OP } z \Rightarrow \text{errore di sintassi}$

- È possibile definire anche la precedenza, attraverso l'ordine delle dichiarazioni

`%right '='`

`%left '+' '-'`

`%left '*' '/'`

-
Priorità
↓
+

Codice User-defined

- È **necessario** definire due metodi
 - `void yyerror(String msg)`: permette di definire come trattare i messaggi di errore
 - `int yylex()`: usato dal *parser* per ottenere i *token* identificati dal *lexer*
- È possibile definire altri metodi e campi, che verranno inclusi nella classe generata
 - Costruttori personalizzati della classe `Parser`
 - Variabili di supporto
 - Il metodo `main`
 - Deve invocare il metodo `yyparse()` della classe `Parser`

Codice User-defined

- Tutto ciò che è tra %{ e %} viene ignorato da BYACC/J. Quindi può essere usato per inserire definizioni Java, include e import di librerie.
- Ad esempio:

```
%{  
import java.lang.Math; import java.io.*;  
import java.util.StringTokenizer;  
%}  
/* YACC Declarations */  
%token NUM  
%left '-' '+'  
%left '*' '/'  
%left NEG /* negation--unary minus */  
%right '^' /* exponentiation */  
...
```

Come eseguire BYACC/J

YACC E JAVA

BYACC/J: Parametri

- Obbligatori
 - `-J`: **deve** essere specificato **esplicitamente** per fare generare codice Java a BYACC/J
 - `<input_file>`: contiene la definizione della grammatica
- Opzionali ¹
 - `-Jclass=<nome_classe>`: permette di specificare il nome della classe generata (e del file `.java` che la contiene)
 - `-Jpackage=<nome_package>`: permette di specificare il package della classe generata
 - `-Jthrows=<lista_eccezioni>`: permette di specificare quali eccezioni possono essere «rilanciate» dal metodo `yparse()`

1. Specificare almeno uno di questi parametri rende non necessario l'uso del parametro obbligatorio `-J`.

BYACC/J: Operazioni Preliminari

1. Estrazione/Installazione di BYACC/J
(scegliamo i binari per Windows)
 - Scompattare il file `.zip`, es., nella cartella
`C:\BYACCJ`
- Non necessita di ulteriori accorgimenti per poter essere eseguito
 - L'archivio `.zip` contiene solamente un file eseguibile (`yacc.exe`)

BYACC/J: Esempio d'Uso

1. Creazione di un file `esempio.y`¹
2. Esecuzione del comando

```
yacc -J esempio.y
```

3. Compilazione delle classi Java nella directory di installazione di BYACC/J
 - Il «classico» `javac *.java`
 - (vedi tip)

1. Per semplicità il file verrà creato o copiato nella cartella di installazione di BYACC/J.

FLEX E JAVA

Flex e Java

- JFlex – The Fast Scanner Generator for Java
 - È compatibile con la sintassi di LeX
 - È scritto in Java e produce codice Java

- Per eseguire JFlex, dalla riga di comando¹:

```
jflex <options> <inputfiles>
```

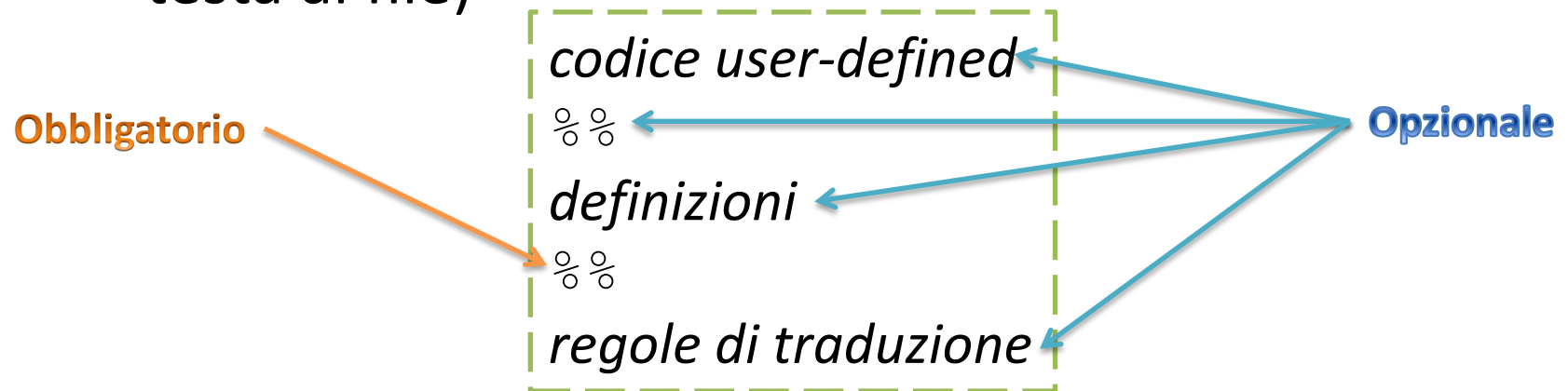
oppure

```
java JFlex.Main <options> <inputfiles>
```

1. Da eseguire dopo essersi posizionati nella cartella `bin` all'interno della cartella di installazione di JFlex.

Struttura dei File `.flex`

- Differisce leggermente dalla struttura dei file `.lex` per il tool LeX
 - Presenta le stesse tre sezioni di dichiarazione, **ma** in un ordine diverso (il codice *user defined* è in testa al file)



Un file di input di JFlex contenente solo `%%` genera un lexer che copia tutto il testo preso in input sull'output...

Codice *User-defined*

- Viene copiato in blocco all'inizio del file `.java` generato
- In questa sezione vengono dichiarate le direttive `package` e `import`

```
package it.unimib.langecomp;  
  
import java.io.*;  
import java.math.*;  
...
```

- Si possono definire anche classi di supporto (*helper classes*), ma non è considerata una buona pratica
 - Meglio definirle in file `.java` esterni

Definizioni (1)

- Permettono di personalizzare la classe generata
- Devono obbligatoriamente iniziare con il carattere %
- «Opzioni» della classe
 - `%class "nome_classe"`
 - `%extends "nome_classe"`
 - e**
 - `%implements "nome_interfaccia", ...`
 - `%public, %final e %abstract`

Definizioni (2)

- Codice della classe

- `% {`
 `...`
 `% }` } Codice incluso in blocco all'interno della classe, es., le dichiarazioni dei campi della classe. Se vi sono più dichiarazioni di questo tipo, vengono concatenate.
- `%init {`
 `...`
 `%init }` } Codice incluso in blocco all'interno del costruttore della classe, es., le dichiarazioni dei campi della classe. Se vi sono più dichiarazioni di questo tipo, vengono concatenate.

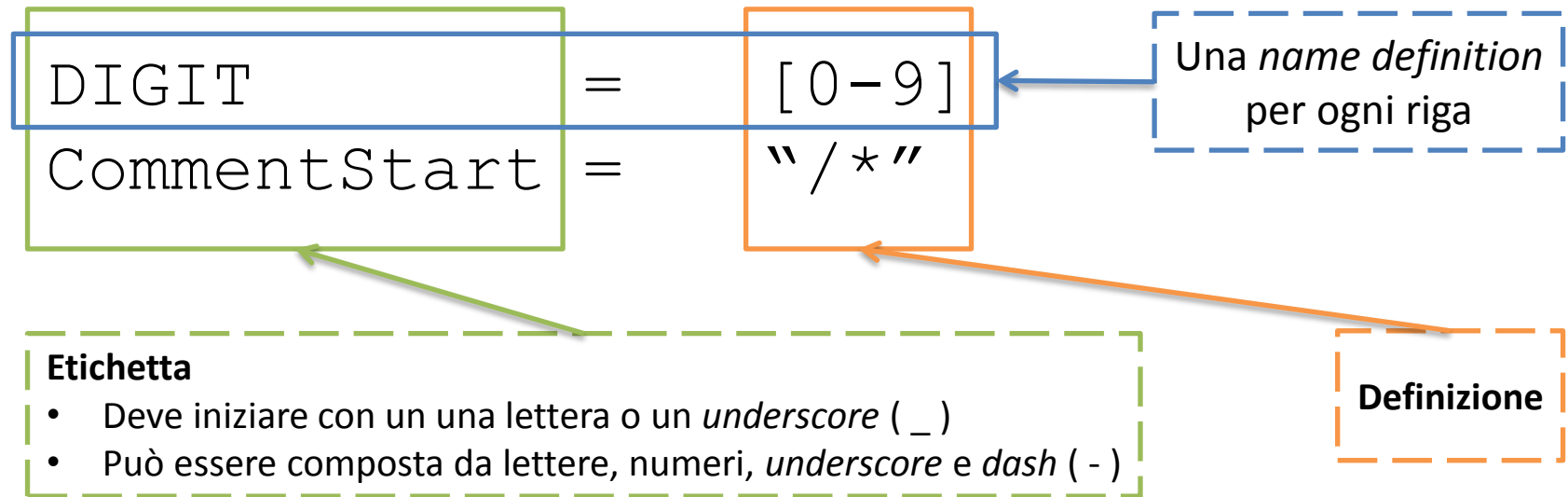
- Compatibilità con BYACC/J

- `%byaccj`

Definizioni: *Name Definitions*

- Chiamate anche *macro*
- Servono per definire una volta sola i *pattern* che si ripetono più volte
 - Anche in diverse regole lessicali

Definizioni: *Name Definitions*



Si riferenziano attraverso la notazione {etichetta}

- Ad esempio, {DIGIT} equivale a [0-9]

Definizioni: *Name Definitions*

Esempio:

- Nella sezione delle definizioni

```
...  
DIGIT = [0-9]  
...
```

- Nella sezione delle regole

```
{DIGIT}+"."{DIGIT}* { System.out.println(...); }
```

equivale a scrivere

```
[0-9]+"." [0-9]* {System.out.println(...); }
```

«Regole di traduzione lessicale»

- Identificano ed eventualmente restituiscono i *token*



Strategia di risoluzione dei pattern:

1. Viene scelto il più lungo tra i pattern verificati
2. Se più pattern con la stessa lunghezza sono verificati, allora viene selezionato il primo
3. Se nessun pattern è verificato, il comportamento di default è di copiare il carattere seguente sullo *standard output*

Definizioni: *Start Conditions*

- E' possibile «pilotare» la valutazione dei *pattern*
- *Come?*
 - Attraverso le *Start Conditions*
- *Cosa sono?*
 - Sono stati di una macchina a stati finiti
- Due tipi
 1. *Inclusive*
 2. *Exclusive*

Definizioni: *Start Conditions*

Nella sezione delle dichiarazioni, si dichiarano nella forma

```
%s list_of_inclusive_start_conditions
```

oppure

```
%x list_of_exclusive_start_conditions
```

Esempio: %s COND1, COND2, COND3, ...

In alternativa possono essere dichiarate con:

- %state (equivalente a %s) □ stati inclusivi
- %xstate (equivalente a %x) □ stati esclusivi

Esiste uno stato predefinito YYINITIAL dichiarato implicitamente

Definizioni: *Start Conditions*

Nella sezione delle regole, si referenziano attraverso l'operatore `<nomi_start_conditions>` anteposto al *pattern* di una regola, es.:

```
<COND2, COND3>[0-9]* {System.out.println(...);}
```

In alternativa, è possibile associare più regole lessicali allo stesso stato attraverso la sintassi compatta, ad esempio:

```
<COND1, COND2> {  
    [0-9]* {System.out.println(...);}  
    [a-z]+ {System.out.print (...);}  
}
```


Definizioni: *Start Conditions*

- Una regola può avere una o più *start conditions* ($\langle \text{COND1}, \text{COND3} \rangle$) oppure nessuna
 - Una regola senza *start conditions* è associata implicitamente allo stato `YYINITIAL`
- Le regole che non specificano alcuna *start condition* vengono utilizzate quando la condizione attiva (stato) è *inclusive*
 - `YYINITIAL` è uno stato *inclusive*
- Se la condizione attiva è *exclusive*, allora solo le regole a cui è associata esplicitamente vengono eseguite

Classe Generata: Metodi Principali

- `int yylex()`: invocato per processare la stringa di caratteri in input al *lexer*
- `String yytext()`: restituisce la stringa che verifica l'espressione regolare
- `int yylength()`: restituisce la lunghezza della stringa che verifica l'espressione regolare
- `char yycharat(int pos)`: restituisce il carattere in posizione `pos` nella stringa che verifica l'espressione regolare; è equivalente a `yytext().charAt(pos)`, ma più veloce. `pos` deve essere compreso tra 0 e `yylength() - 1`.
- `void yyclose()`: chiude lo stream di input
- `int yystate()`: restituisce lo stato corrente del *lexer*
- `void yybegin(int lexicalState)`: pone il *lexer* nello stato `lexicalState`

Come eseguire JFlex

FLEX E JAVA

JFlex: Parametri

- Obbligatori
 - `<inputfiles>`: uno (o più) file `.flex` da cui generare il codice del lexer
es., `file_1.flex file_2.flex ... file_n.flex`
- Facoltativi (i più significativi)
 - `-d <dir>`: memorizza i file `.java` in `dir`
 - `--verbose` o `-v`: mostra tutti i messaggi
 - `--quiet` or `-q`: mostra solo i messaggi di errore

JFlex: Operazioni Preliminari

1. Estrazione/Installazione di JFlex (scegliamo la piattaforma Windows)
 - Scompattare il file `.zip`, es., nella cartella `C:\JFlex`
2. Impostazione parametri di esecuzione
 - Nella sottocartella `bin` in `C:\JFlex`
 1. Editare il file `jflex.bat`
 2. Impostare la variabile `JFLEX_HOME` a `C:\JFlex`
 3. Impostare la variabile `JAVA_HOME` al percorso che punta alla cartella di installazione del JDK

JFlex: Esempio d'Uso (1)

- Creazione del file `esempio.flex`¹
- Esecuzione del comando

```
jflex -d C:\out_dir --verbose esempio.flex
```

oppure

```
java JFlex.Main -d C:\out_dir -v esempio.flex
```

- Compilazione delle classi Java nella directory
`C:\out_dir`
 - Il «classico» `javac *.java`

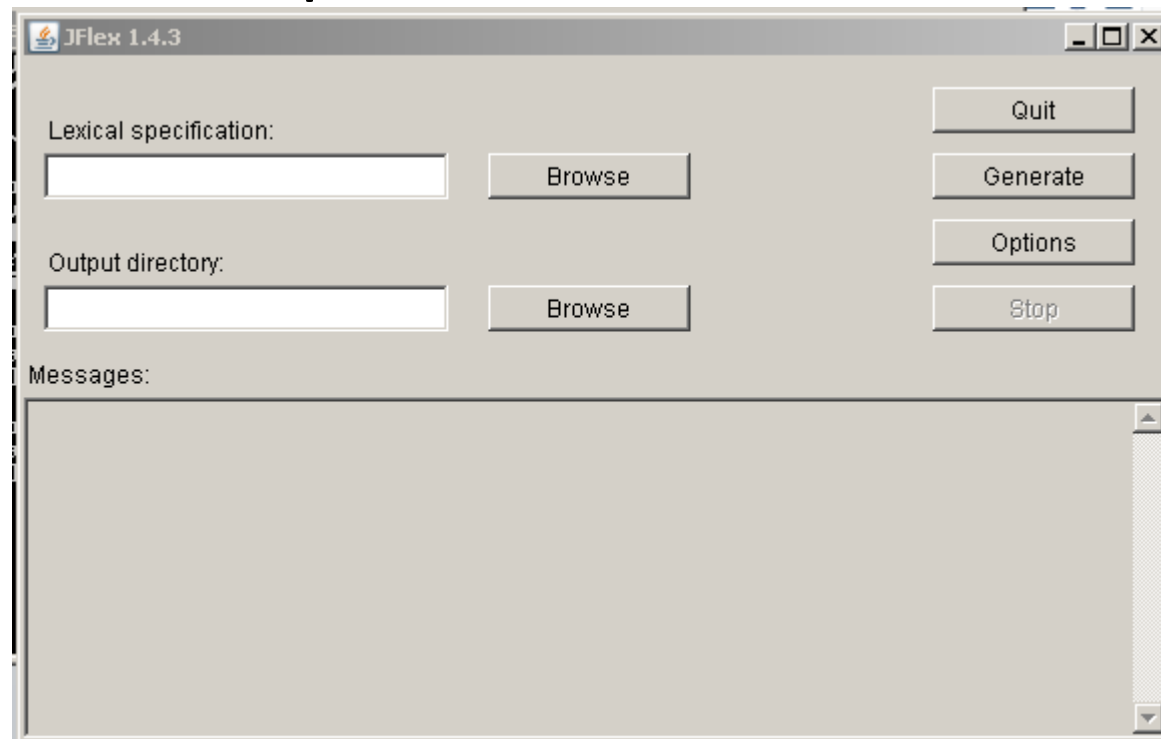
1. Per semplicità il file verrà creato o copiato nella cartella `bin` all'interno della cartella di installazione di JFlex.

JFlex: Esempio d'Uso (2)

- Esecuzione del comando

`jflex`

apre una semplice GUI



Le Espressioni Regolari

FLEX E JAVA

Lex & Espressioni Regolari

- I pattern delle regole di JFlex si basano sulle espressioni regolari per riconoscere le sequenze di caratteri in input (*pattern matching*)
- Le espressioni regolari si basano sul formato standard utilizzato dai *tool* disponibili in ambiente POSIX (es.: s.o. UNIX quale `grep`)

Pattern Semplici

Pattern	Descrizione
x	Il carattere x (minuscolo)
X	Il carattere X (maiuscolo)
[amz]	Il carattere a, m o z
[0-9]	Un carattere compreso tra 0 e 9
[a-z]	Un carattere compreso tra a e z (in minuscolo e inclusi gli estremi, cioè a e z)
[A-Z]	Un carattere compreso tra A e Z (in maiuscolo e inclusi gli estremi, cioè A e Z)
[abd-zZ]	Un carattere minuscolo compreso tra a a z, escluso c, più Z (maiuscolo)
[a-zA-Z]	Un carattere compreso tra a e z oppure tra A e Z
.	Un carattere qualsiasi
\n	Andata a capo

Pattern Complessi (1)

Pattern ¹	Descrizione
[^r]	Classe di caratteri negata: identifica un carattere che non corrisponde al pattern specificato dopo [^; es., [^a-c] è verificato per qualsiasi carattere che non sia a, b o c.
r?	Il pattern può non comparire o comparire una sola volta.
r*	Il pattern può comparire zero o più volte.
r+	Il pattern può comparire una o più volte.
r{n}	Il pattern deve ripetersi n volte.
r{n, m}	Il pattern può ripetersi da minimo n volte a massimo m volte.
r{n, }	Il pattern deve ripetersi minimo n volte; nessun limite superiore.

1. Con r ed s sono indicati dei pattern (espressioni regolari) generici, es., [0-9] o [a-zA-Z].

Pattern Complessi (2)

Pattern ¹	Descrizione
rs	Il pattern r deve essere seguito dal pattern s .
$r s$	Il pattern r oppure il pattern s .
r	Il pattern r viene «cercato» solo nella parte iniziale della stringa di caratteri
$r\$$	Il pattern r viene «cercato» solo nella parte finale della stringa di caratteri
(r)	Crea un gruppo contenente il pattern r

1. Con r ed s sono indicati dei pattern (espressioni regolari) generici, es., $[0-9]$ o $[a-zA-Z]$.

Caratteri di *Escape*

- Alcuni caratteri hanno un significato speciale nelle espressioni regolari, altri nel testo fornito in input al lexer
 - Non possono essere «usati» direttamente
 - Bisogna effettuare l'*escape*: si antepone il carattere `\` al carattere

Pattern	Descrizione
<code>\a, \b, \f, \n, \r, \t, \v</code>	I caratteri speciali delle stringhe ANSI-C.
<code>*, \", \', \+, \?, \[, ...</code>	I caratteri riservati delle espressioni regolari.
<code>\123, \x2a</code>	Il carattere con valore ottale 123 e il carattere con valore esadecimale 2a.

«Estensioni» di JFlex

- I patter di Lex possono essere definiti usando anche costrutti «non standard»

Pattern ¹	Descrizione
{nome_definizione}	Richiama una <i>name definition</i> e la include nel pattern.
r/s	Il pattern (composto) è verificato se r segue il pattern s , ma solo la stringa che verifica r viene considerata l'input corrente.
"[a]\"foo"	Considera tutti i caratteri racchiusi tra " come un pattern letterale.

1. Con r ed s sono indicati dei pattern (espressioni regolari) generici, es., $[0-9]$ o $[a-zA-Z]$.

Qualche Esempio (1)

Pattern	Stringa Valida	Stringa NON Valida
<code>[0-9]+</code>	<ul style="list-style-type: none"> • 123 • a123 	<ul style="list-style-type: none"> • aaabaa
<code>[a-z]+</code>	<ul style="list-style-type: none"> • prova 	<ul style="list-style-type: none"> • 123
<code>[A-Z]?</code>	<ul style="list-style-type: none"> • A • 1A • A123 • 1 	
<code>^[A-Z]+</code>	<ul style="list-style-type: none"> • A • A1 • AA • AA1 	<ul style="list-style-type: none"> • 1A
<code>[A-Z]+\$</code>	<ul style="list-style-type: none"> • 1A • 1AA 	<ul style="list-style-type: none"> • A1 • 1a
<code>a{3}b</code>	<ul style="list-style-type: none"> • aaaabaa 	<ul style="list-style-type: none"> • Aabba

Qualche Esempio (2)

Pattern	Stringa Valida	Stringa NON Valida
<code>a{2,4}b</code>	<ul style="list-style-type: none"> • aab • aaab 	<ul style="list-style-type: none"> • abaa
<code>[a-z]+[\t]+[0-9]+</code>	<ul style="list-style-type: none"> • prova 1 • Prova 1 	<ul style="list-style-type: none"> • 123 • prova1
Esempi reali		
<code>(19 20)[0-9]{2}</code> Anni dal 1900 al 2099	<ul style="list-style-type: none"> • 1991 • 2012 	<ul style="list-style-type: none"> • 12 • 2100
<code>0[1-9] 1[012]</code> Mesi in formato numerico a due cifre	<ul style="list-style-type: none"> • 12 	<ul style="list-style-type: none"> • 1
<code>(19 20)[0-9]{2}\\. (0[1-9] 1[012])\\. (0[1-9] 12)[0-9]{3}</code> Data in formato yyyy\\mm\\gg	<ul style="list-style-type: none"> • 2012\\12\\12 	<ul style="list-style-type: none"> • 2012-12-12
<code>^[a-zA-Z0-9-_.]+@[a-zA-Z0-9-]+\\. [a-zA-Z]{2,4}\$</code>	<ul style="list-style-type: none"> • nome.c@e.com 	<ul style="list-style-type: none"> • nome.c@e

Creazione di un Ambiente Integrato per la Generazione di Parser in Java

BYACC/J & JFLEX

Ottenere i Due Tool

- JFlex
 - <http://jflex.de/download.html>
 - Archivio per Windows (.zip), RedHat Linux (.rpm) e generico (.tar.gz)
- BYACC/J
 - <http://byaccj.sourceforge.net/#download>
 - Archivio per Windows (.zip), Linux, MacOS, Solaris e generico (.tar.gz)
- Ma noi faremo riferimento ai file condivisi su Moodle

Setup dell'Ambiente di Esecuzione

Windows *

1. Aprire il file `.zip` condiviso ed estrarre la cartella `jflex-1.4.3` in `C:\` **
2. Spostarsi nella sottocartella `C:\jflex-1.4.3\bin`
3. Editare (tasto destro, ...) il file `jflex.bat`
4. Impostare la variabile `JFLEX_HOME` a `C:\jflex-1.4.3`
5. Impostare la variabile `JAVA_HOME` al percorso che punta alla cartella di installazione del JDK (la più recente che avete sul sistema, ad es.: `C:\Program Files\Java\jdk1.7.0_05`)

*: per un corrispondente tutorial per MacOS X e sistemi UNIX/Linux si vedano le ultime slides .

** : «C» sta per la root in cui potete salvare, può anche essere Z:\

Esecuzione dei Tool

*Windows**

- Utilizzo della riga di comando
 1. Aprire il Prompt dei Comandi (Start □ Esegui □ cmd)
 2. Portarsi nella sottocartella C:\jflex-1.4.3\bin con il comando `cd C:\jflex-1.4.3\bin`
 3. Eseguire in sequenza i due tool
 1. `yacc -J calc.y`
 2. `jflex calc.flex`
 4. Eseguire il compilatore Java
 - `javac *.java`
 5. Eseguire il programma generato
`java -cp . Parser`

Una Semplice Calcolatrice Scritta con BYACC/J e JFlex*

UN SEMPLICE ESEMPIO

*. <http://trac.jmodelica.org/browser/tags/1.0b2/ThirdParty/JFlex/jflex-1.4.3/examples/byaccj>

I File di Definizione

- Guardate due file nella sottocartella
`C:\jflex-1.4.3\bin`
 1. `calc.y`
 2. `calc.flex`
- Sono comuni file di testo
 - Qualsiasi editor di testo un po' evoluto va bene per editarli (notepad è da evitare).

calc.y (1)

```
%{  
    import java.io.*;  
}%  
  
%token NL          /* newline */  
%token <dval> NUM  /* a number */  
  
%type <dval> exp  
  
%left '-' '+'  
%left '*' '/'  
%left NEG          /* negation--unary minus */  
%right '^'         /* exponentiation */  
  
%%
```

calc.y (2)

```
input:    /* empty string */
         | input line
         ;

line:     NL          { if (interactive)
                        System.out.print("Expression: "); }
         | exp NL     { System.out.println(" = " + $1);
                        if (interactive)
                            System.out.print("Expression: "); }
         ;
```


calc.y (3)

```
exp:      NUM                { $$ = $1; }
      | exp '+' exp          { $$ = $1 + $3; }
      | exp '-' exp          { $$ = $1 - $3; }
      | exp '*' exp          { $$ = $1 * $3; }
      | exp '/' exp          { $$ = $1 / $3; }
      | '-' exp %prec NEG    { $$ = -$2; }
      | exp '^' exp          { $$ = Math.pow($1, $3); }
      | '(' exp ')'          { $$ = $2; }
      ;
```

%%

calc.y (4)

```
private Yylex lexer;
private int yylex () {
    int yyl_return = -1;
    try {
        yyval = new ParserVal(0);
        yyl_return = lexer.yylex();
    }
    catch (IOException e) {
        System.err.println("IO error :"+e);
    }
    return yyl_return;
}

public void yyerror (String error) {
    System.err.println ("Error: " + error);
}
```

calc.y (5)

```
public Parser(Reader r) {
    lexer = new Yylex(r, this);
}

static boolean interactive;

public static void main(String args[]) throws IOException {
    System.out.println("BYACC/Java with JFlex Calculator Demo");
    Parser yyparser;
    if ( args.length > 0 ) {
        // parse a file
        yyparser = new Parser(new FileReader(args[0]));
    }
    else {
        // interactive mode
        System.out.println("[Quit with CTRL-D]");
    }
}
```

calc.y (6)

```
System.out.print("Expression: ");
interactive = true;
yyparser = new Parser(new InputStreamReader(System.in));
}
yyparser.yyparse();
if (interactive) {
    System.out.println();
    System.out.println("Have a nice day");
}
}
```

calc.flex (1)

%%

%byaccj

{

private Parser yyparser;

public Yylex(java.io.Reader r, Parser yyparser) {

this(r);

this.yyparser = yyparser;

}

}

NUM = [0-9]+ ("." [0-9]+) ?

NL = \n | \r | \r\n

%%

calc.flex (2)

```
/* operators */
"+" |
"-" |
"*" |
"/" |
"^" |
"(" |
")" { return (int) ycharat(0); }

/* newline */
{NL} { return Parser.NL; }

/* float */
{NUM} { yyparser.yylval =
        new ParserVal(Double.parseDouble(yytext()));
        return Parser.NUM; }
```

calc.flex (3)

```
/* whitespace */
[ \t]+ { }\b      { System.err.println("Sorry, backspace doesn't
work"); }

/* error fallback */
[^]      { System.err.println("Error: unexpected character
                                   '"+yytext()+"'");
          return -1; }
```

Compilazione (1)

- Dalla riga di comando
 1. Portarsi nella sottocartella `C:\jflex-1.4.3\bin` con il comando
`cd C:\jflex-1.4.3\bin`
 2. Eseguire in sequenza i due tool
 1. `yacc -J calc.y`
 2. `jflex calc.flex`
 3. Eseguire il compilatore Java
 - `javac *.java`

Compilazione (2)

```
C:\Users\Iade\Desktop\JParser\bin>jflex calc.flex
Reading "calc.flex"
Constructing NFA : 42 states in NFA
Converting NFA to DFA :
.....
20 states before minimization, 10 states in minimized DFA
Writing code to "Yylex.java"

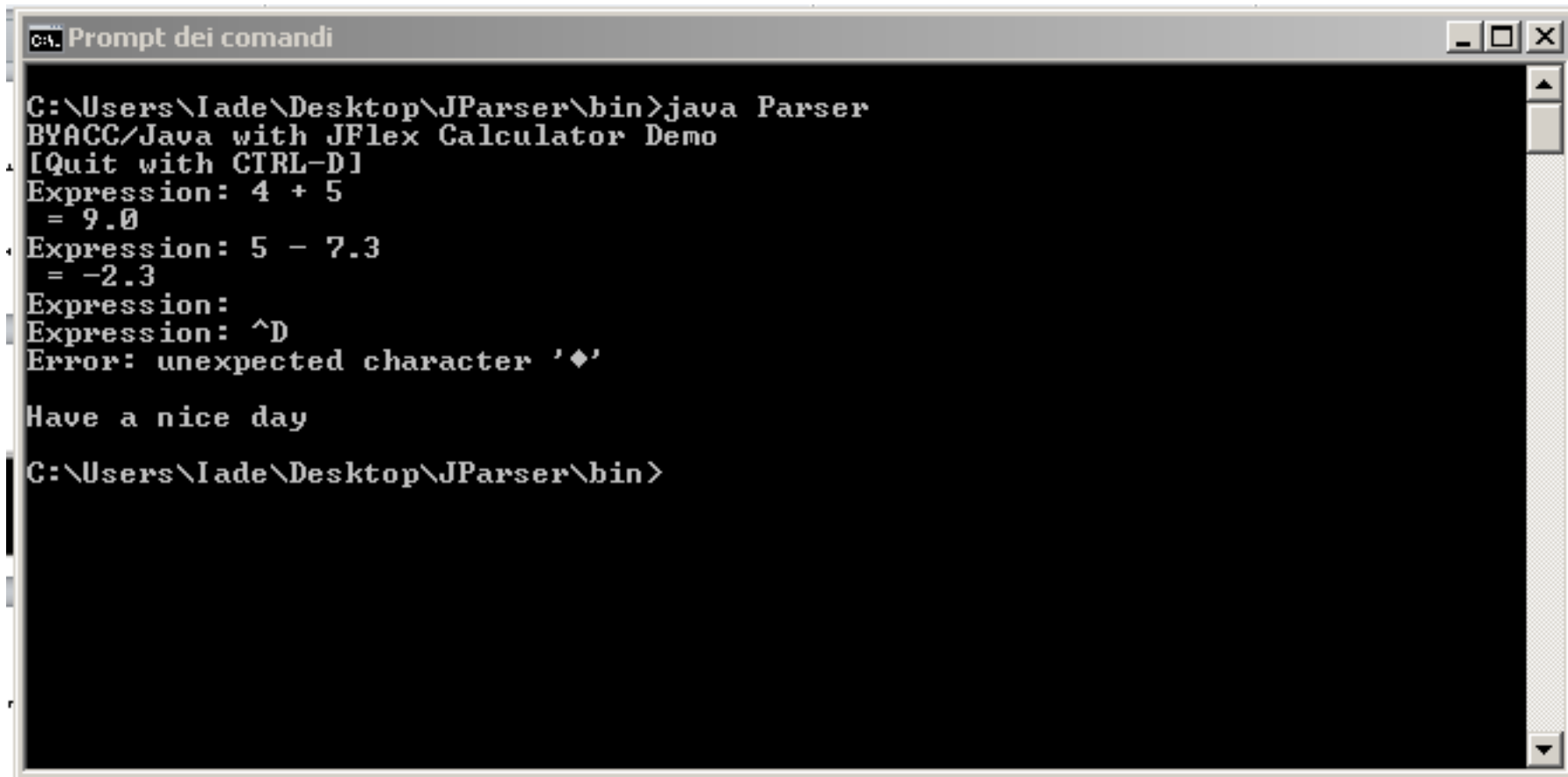
C:\Users\Iade\Desktop\JParser\bin>
```

```
C:\Users\Iade\Desktop\JParser\bin>yacc -J calc.y
C:\Users\Iade\Desktop\JParser\bin>javac *.java
C:\Users\Iade\Desktop\JParser\bin>
```

Esecuzione (1)

- (Valido solo se nel codice user-defined è stato definito il metodo `main`)
- Dalla riga di comando
 1. Portarsi nella sottocartella `C:\jflex-1.4.3\bin` con il comando
`cd C:\jflex-1.4.3\bin`
 2. Eseguire il programma generato
 - `java -cp . Parser`

Esecuzione (2)



```
C:\Users\Iade\Desktop\JParser\bin>java Parser
BYACC/Java with JFlex Calculator Demo
[Quit with CTRL-D]
Expression: 4 + 5
= 9.0
Expression: 5 - 7.3
= -2.3
Expression:
Expression: ^D
Error: unexpected character '^'
Have a nice day
C:\Users\Iade\Desktop\JParser\bin>
```

Risorse/Documentazione

- JFlex
 - <http://jflex.de/manual.html>
- BYACC/J
 - <http://byaccj.sourceforge.net/>
- Altra documentazione
 - LeX (originale) □ Descrizione dettagliata della sintassi di LeX, da cui deriva quella di JFlex
<http://dinosaur.compilertools.net/lex/index.html>
 - Yacc (originale) □ Descrizione dettagliata della sintassi di Yacc, da cui deriva quella di BYACC/J
<http://dinosaur.compilertools.net/yacc/index.html>
 - Espressioni regolari □ Descrizione della sintassi UNIX per definire i pattern, corredata di esempi
<http://www.regular-expressions.info/tutorial.html>

Appendice

SETUP ED ESECUZIONE DEI TOOL PER AMBIENTI NON-WINDOWS

Setup dell'Ambiente di Esecuzione

MacOS X

1. Aprire il file `.zip` di JFlex ed estrarre la cartella `jflex-1.4.3` nella «cartella» Applicazioni
2. Aprire il file `.zip` di BYACC/J ed estrarre il file `yacc.macosx` nella sottocartella `jflex-1.4.3/bin` nella «cartella» Applicazioni

Esecuzione dei Tool

MacOS X

- Utilizzo della riga di comando
 1. Aprire il Terminale
 2. Portarsi nella sottocartella `jflex-1.4.3/bin` nella «cartella» Applicazioni
 3. Eseguire in sequenza i due tool
 1. `./yacc.macosx -J calc.y`
 2. `./jflex calc.flex`
 4. Eseguire il compilatore Java
 - `javac *.java`
 5. Eseguire il programma generato (se si è definito un `main`)
 - `java Parser`

Setup dell'Ambiente di Esecuzione

UNIX/Linux

Installare i pacchetti `byaccj` e `jflex` forniti con la propria distribuzione utilizzando il package manager (es., *Ubuntu Software Center*, *Synaptic*, ecc.)

Esecuzione dei Tool

UNIX/Linux

- Utilizzo della riga di comando
 1. Aprire il Terminale
 2. Portarsi nella cartella contenente i file `.y` e `.flex`
 3. Eseguire in sequenza i due tool
 1. `byaccj -J calc.y`
 2. `jflex calc.flex`
 4. Eseguire il compilatore Java
 - `javac *.java`
 5. Eseguire il programma generato (se si è definito un `main`)
 - `java Parser`

Alcuni suggerimenti (tips)

1) Se il comando 'java Parser' restituisce l'errore "Errore: Impossibile trovare o caricare la classe principale Parser" o simile, è possibile risolvere il problema con il seguente comando:

```
java -cp . Parser
```

che imposta il CLASSPATH alla carella corrente.

Alcuni suggerimenti (tips)

2) E' possibile salvare l'output dell'esecuzione del proprio parser senza ricorrere a classi particolari del package java.io (es. FileWriter e simili); per ottenere il medesimo risultato, è sufficiente usare le opportune chiamate a System.out.print() e System.out.println() nelle azioni ed eseguire il parser con la seguente riga di comando

```
java -cp . Parser nome_file.txt > output_file.txt
```

Il metodo più rapido per fare debug con il codice generato da byacc/j è inserire delle chiamate al metodo System.out.println() per stampare su standard output dei messaggi (che potrà rimuovere una volta terminato il debug del codice).

Alcuni suggerimenti (tips)

3) Per poter permettere al parser, generato con BYACC/J, di prendere in input un file di testo, è sufficiente specificare il nome del file come parametro della riga di comando:

```
java -cp . Parser nome_file.txt
```

In questo modo il contenuto del file verrà utilizzato come stringa input per il parser.