JavaCC - Generalità

- **che cosa fa**: partendo da una specifica di una grammatica (con azioni semantiche), genera le classi Java che realizzano un analizzatore sintattico top-down per tale grammatica
- **specifica JavaCC** file testuale con estensione `.jj` suddiviso in tre sezioni:
  1. lista di opzioni per JavaCC
  2. unità di compilazione Java che inizia con `PARSER_BEGIN(nome_parser)` e termina con `PARSER_END(nome_parser)`
  3. lista di regole lessicali e sintattiche (ovvero produzioni poste essenzialmente in grammatica EBNF)
- si possono aggiungere **azioni semantiche**, ovvero, porzioni di codice Java eseguite al momento dell'espansione delle produzioni

Specifica JavaCC: esempio

esempio: specifica JavaCC che implementa un *riconoscitore* per la seguente grammatica per espressioni aritmetiche:

$$
\begin{aligned}
&S \rightarrow E \ \textit{eof} \\
&E \rightarrow T \ E_1 \\
&E_1 \rightarrow + \ T \ E_1 \mid \epsilon \\
&T \rightarrow F \ T_1 \\
&T_1 \rightarrow * \ F \ T_1 \mid \epsilon \\
&F \rightarrow (\ E \ ) \mid \textit{num}
\end{aligned}
$$

(dove *num* denota le costanti intere senza segno e *eof* denota il simbolo di end-of-file)

Specifica JavaCC 1: lista di opzioni per JavaCC

Sezione posta all'inizio della specifica JavaCC: il compilatore assegna alle opzioni non specificate un valore di default

Risulta quindi necessario inserire solo le opzioni che si intende modificare.

```
/* file expr1.jj - riconoscitore di espressioni aritmetiche */

OPTION
{
    STATIC           = false;     // default TRUE
    LOOKAHEAD        = 2;         // default 1
    IGNORE_CASE      = true;      // default false
    OUTPUT_DIRECTORY = "path";    // default directory corrente
}
```

Specifica JavaCC 2: dichiarazione classe parser

```
PARSER_BEGIN(expr1)

// eventuale dichiarazione di package

// eventuale lista di import

public class expr1 {

  public static void main(String args[]) throws ParseException {
    expr1 parser = new expr1(System.in);
    parser.start();
  }
}

PARSER_END(expr1)
```

Specifica JavaCC 3.1: regole lessicali

Sono suddivise in quattro stati lessicali: SKIP, TOKEN, MORE, SPECIAL_TOKEN

```
SKIP : { " " | "\t" | "\n" | "\r" }

TOKEN :
{
    <NUM: (["0"-"9"])+>
  | <PIU: "+">
  | <MENO: "-">
  | <PER: "*">
  | <DIV: "/">
  | <PARAP: "(">
  | <PARCH: ")">
}
```

(N.B.: <EOF> è un token predefinito in JavaCC)

Specifica JavaCC 3.2: regole sintattiche

```
void start() : {}
{ espr() <EOF> }

void espr() : {}
{ term() espr1() }

void espr1() : {}
{  <PIU> term() espr1() | <MENO> term() espr1() | {} }

void term() : {}
{ factor() term1() }

void term1() : {}
{  <PER> factor() term1() | <DIV> factor() term1() | {} }

void factor() : {}
{ <PARAP> espr() <PARCH> | <NUM> }
```

File prodotti da JavaCC

Tramite il comando "`javacc expr1.jj`" vengono prodotte le seguenti classi:

- **classi generiche** (generate solo se non presenti):

  `SimpleCharStream.java`

  `Token.java`

  `ParseException.java`

  `TokenMgrError.java`

- **classi specifiche** (generate ad ogni esecuzione di javacc):

  `nome_parser.java`

  `nome_parserConstants.java`

  `nome_parserTokenManager.java`

Che vanno poi compilate mediante il comando "`javac *.java`"

Domanda: **È possibile modificare le classi prodotte da JavaCC?**

Analisi lessicale: il token manager

- implementato dalla classe `nome_parserTokenManager.java`
- analizza il flusso di caratteri in ingresso suddividendolo in porzioni (Token) in base alla specifica lessicale contenuta nel file `.jj`
- istanzia un oggetto della classe `Token.java` per ogni porzione di testo in ingresso individuata
- associa ad ogni oggetto istanziato alcune informazioni come il tipo (`kind`) e la corrispondente porzione di testo ad esso associata (`image`)
- tali informazioni possono essere accedute come attributi dell'oggetto Token
- collabora con il parser per creare la sequenza di Token utilizzando le regole di produzione presenti nella specifica JavaCC

Analisi sintattica: il parser

- implementato dalla classe `nome_parser.java`
- contiene un metodo (funzione) per ogni simbolo non terminale presente nella specifica della grammatica
- a partire dal metodo relativo all'assioma (specificato nella dichiarazione della classe parser della specifica JavaCC) applica le produzioni della grammatica invocando i corrispondenti metodi
- strategia **top-down**: dall'assioma ai terminali
- decisioni sintattiche
- JavaCC non applica la tecnica del backtracking - problemi di performance

Analisi sintattica (2): punti di decisione

Sono punti in cui il parser deve “decidere” in che modo continuare l’analisi sintattica dell’input

possibili punti di scelta

- `( prod_1 | prod_2 | ... | prod_N )`
- `( prod )?` oppure `[ prod ]`
- `( prod )*`
- `( prod )+`

Algoritmo di scelta: il parser “guarda in avanti” il prossimo Token e prende una decisione in base al Token analizzato, senza fare backtracking

Tale algoritmo è applicabile solo se la grammatica LL(1).

Analisi sintattica (2): il LOOKAHEAD

È possibile modificare l'algoritmo di scelta di default, indicando al parser un valore di `LOOKAHEAD` diverso da quello predefinito (1). Abbiamo due alternative:

- impostare un valore di `LOOKAHEAD` **globale**, specificandone il valore nella sezione di dichiarazione delle opzioni (prima parte della specifica JavaCC - opzione `LOOKAHEAD`)
- impostare un valore di `LOOKAHEAD` **locale**, specificandolo subito prima del punto di decisione (miglioramento di performance)

Esempio: `( LOOKAHEAD(2) prod_1 | prod_2 )`

N.B.: al momento della compilazione JavaCC produce un messaggio di WARNING nel caso ci siano delle ambiguità che non riesce a risolvere con il valore di `LOOKAHEAD` di default.

Uso delle azioni semantiche

- vogliamo estendere il riconoscitore generato dalla precedente specifica
- lo scopo è quello di stampare in uscita la sequenza di regole di produzione che genera la derivazione canonica sinistra della stringa di input
- aggiungiamo pertanto alla precedente specifica delle semplici azioni semantiche
- in particolare, *all'inizio* di ogni regola di produzione aggiungiamo una istruzione di stampa

Specifica JavaCC 5: uso delle azioni semantiche

```
void start() : {}
{ {System.out.println("S -> E eof");} term() espr1() }

void espr() : {}
{ {System.out.println("E -> T E1");} term() espr1() }

void espr1() : {}
{  {System.out.println("E1 -> '+' T E1");} <PIU> term() espr1()
 | {System.out.println("E1 -> '-' T E1");} <MENO> term() espr1()
 | {System.out.println("E1 -> epsilon");} }

void term() : {}
{ { System.out.println("T -> F T1"); } factor() term1() }

void term1() : {}
{  {System.out.println("T1 -> '*' F T1");} <PER> factor() term1()
 | {System.out.println("T1 -> '/' F T1");} <DIV> factor() term1()
 | {System.out.println("T1 -> epsilon");} }

void factor() : {}
{  {System.out.println("F -> '(' E ')'");} <PARAP> espr() <PARCH>
 | {System.out.println("F -> num");} <NUM> }
```

Uso degli attributi

- vogliamo ora realizzare con JavaCC un programma che effettua il calcolo del valore dell'espressione aritmetica
- a tale scopo, utilizziamo attributi che associamo ai simboli non terminali e terminali della grammatica
- estensione della grammatica per espressioni aritmetiche con attributi e azioni semantiche per il calcolo del valore dell'espressione
- gli attributi sono realizzati:
  1. definendo *argomenti* nelle funzioni che implementano i simboli non terminali;
  2. facendo *restituire un valore* a tali funzioni

Specifica JavaCC 6: uso degli attributi

```
void start() : {int ris;}
{ ris=espr() <EOF> {System.out.println(ris);} }

int espr() : {int t1,ris;}
{ t1=term() ris=espr1(t1) {return ris;} }

int espr1(int t1) : {int t2,t3,ris;}
{  <PIU> t2=term() {t3=t1+t2;} ris=espr1(t3) {return ris;}
 | <MENO> t2=term() {t3=t1-t2;} ris=espr1(t3) {return ris;}
 | {return t1;}
}
```

Specifica JavaCC 6: uso degli attributi (segue)

```
int term() : {int t1,ris;}
{ t1=factor() ris=term1(t1) {return ris;} }

int term1(int t1) : {int t2,t3,ris;}
{  <PER> t2=factor() {t3=t1*t2;} ris=term1(t3) {return ris;}
 | <DIV> t2=factor() {t3=t1/t2;} ris=term1(t3) {return ris;}
 | {return t1;}
}

int factor() : {int ris; Token n;}
{  <PARAP> ris=espr() <PARCH> {return ris;}
 | n=<NUM> {ris=Integer.parseInt(n.image); return ris;}
}
```

Attributi e simboli terminali

nell'esempio precedente, viene associato un attributo di tipo intero al tipo di token <NUM>

```
int factor() : {int ris; Token n;}
{  <PARAP> ris=espr() <PARCH> {return ris;}
 | n=<NUM> {ris=Integer.parseInt(n.image); return ris;}
}
```

- `n.image` contiene il valore (stringa) del token di tipo <NUM>
- `Integer.parseInt` converte la stringa in intero
- nella variabile `ris` viene memorizzato il valore intero corrispondente al valore del token <NUM>
- tale valore viene restituito in uscita dal metodo `factor`