ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN INGEGNERIA E
SCIENZE INFORMATICHE

TITOLO DELLA RELAZIONE FINALE

STRUMENTI E FRAMEWORK PER LO SVILUPPO DI LINGUAGGI DI PROGRAMMAZIONE E LINGUAGGI DOMAIN-SPECIFIC IN JAVA

Relazione finale in
Sviluppo di linguaggi di programmazione

Relatore
Chiar.mo Prof.
Mirko Viroli

Presentata da
Andrea Sperandio
Matr.: 0000652155

Sessione II
Anno Accademico 2014/2015

Indice

1. Introduzione

Il seguente scritto ha lo scopo di indagare le tecnologie disponibili per la realizzazione di linguaggi di programmazione e linguaggi domain specific in ambiente Java. In particolare, vengono proposti e analizzati tre strumenti presenti sul mercato: JavaCC, ANTLR e Xtext.

I primi due sono generatori di lexers e parsers che permettono di scrivere la grammatica per un certo linguaggio di programmazione, di stabilirne le regole per il controllo lessicale, sintattico e semantico e di definirne il comportamento in base al codice fornito in input. Xtext, invece, è un tool molto più potente, che permette la definizione di nuovi linguaggi di programmazione forniti di compilatore e supportati da funzionalità runtime che aiutano il programmatore nella produzione di codice (come formattatore di codice, code coloring, code completion, etc.).

Al termine dell'elaborato, il lettore dovrebbe avere un'idea generale dei principali meccanismi e sistemi utilizzati (come lexer, parser, AST, parse trees, etc.), oltre che del funzionamento dei tre tools presentati. Inoltre, si vogliono individuare vantaggi e svantaggi di ciascuno strumento attraverso un'analisi delle funzionalità offerte, così da fornire un giudizio critico per la scelta e la valutazione dei sistemi da utilizzare.

Dopo aver introdotto alcuni concetti fondamentali necessari alla comprensione delle varie rassegne, il testo dedica un capitolo per ciascun tool, in cui ne presenta le caratteristiche principali e alcuni meccanismi di dettaglio, per poi concludere con una discussione e un confronto dei diversi aspetti e servizi messi a disposizione da questi strumenti.

Il capitolo “Nozioni fondamentali” introduce concetti come lexer e parser, notazioni (BNF), strutture dati (parser trees e ASTs) e tecniche di funzionamento (come i lookaheads) che sono effettivamente utilizzati dai tools presentati.

Nei capitoli dedicati alle diverse tecnologie, si fornisce un’idea di che cosa è e di come viene impiegato il tool in questione (Introduzione), per poi analizzare le classi e i files che utilizza e che produce durante il suo funzionamento e gli altri linguaggi con cui può eventualmente interagire (Funzionamento).

Il sottocapitolo “Dettagli e modi d’uso” è dedicato alla presentazione approfondita di aspetti riguardanti lo strumento in analisi, come i meccanismi che utilizza (es. lookahead), le keywords che mette a disposizione e altri tool con cui interagisce o su cui si appoggia.

Inoltre viene fornito un esempio concreto realizzato con la tecnologia presentata, in cui viene spiegato passo passo il codice necessario alla realizzazione di un determinato compito.

Infine, il capitolo dedicato alle conclusioni propone giudizi critici sui vari sistemi e approcci presentati, cercando di mettere in evidenza i fattori fondamentali utilizzati per confrontare tra loro le diverse tecnologie.

Ad es. alcuni strumenti sono meglio documentati, altri mettono a disposizione funzionalità specifiche più comode, o si integrano meglio con certi ambienti di sviluppo, o ancora, vengono riportate le ultime date di release per avere un'idea concreta di quanto il prodotto sia attivamente sviluppato, etc..

1.1 Grammatiche e motivazioni

Si pensi ad un problema reale risolubile con l'ausilio di computers: ad esempio la gestione di pratiche di una compagnia assicurativa. La situazione potrebbe essere articolata per l'utilizzo di formalismi interni all'azienda da parte degli impiegati, oppure perché sono applicate business rules particolari nel dominio specifico.

Il problema può essere risolto producendo un software di gestione che rispetti le specifiche; tale prodotto può essere realizzato in un qualsiasi linguaggio di programmazione.

Esiste anche la possibilità, però, di scrivere un linguaggio ad hoc per l'azienda che rispetti la sintassi dettata dai committenti. Per formalizzare tale sintassi occorre una grammatica che definisca precisamente gli elementi, gli operatori e i loro casi di utilizzo all'interno del linguaggio.

Una grammatica formale è una struttura astratta che descrive un linguaggio formale in modo preciso: è un sistema di regole che delineano matematicamente un insieme (di solito infinito) di sequenze finite di simboli (stringhe) appartenenti ad un alfabeto anch'esso finito. La grammatica per un linguaggio di programmazione descrivere le regole che ne definiscono la struttura sintattica.

I linguaggi di programmazione possono essere domain specific o general purpose. Un linguaggio domain specific (DSL) è un piccolo linguaggio di programmazione che si focalizza su un particolare dominio, cosicché i suoi concetti e le sue notazioni siano il più vicino possibile alla soluzione di problemi in quel dominio. Al contrario, un linguaggio general purpose (GPL), come Java, può essere utilizzato per risolvere un ampio range di problemi, ma potrebbe non fornire la migliore strategia di risoluzione.

Nello scrivere un DSL occorre scegliere una sintassi concisa e precisa, che sia facilmente riconoscibile ed utilizzabile dall'utente.

Alcuni esempi di DSL sono rappresentati da SQL, che si focalizza sull'interrogazione di databases relazionali, dalle regular expressions, da CSS, usato per definire la formattazione di documenti HTML, XHTML e XML, o anche da linguaggi forniti da tools come MatLab. Anche la maggior parte dei linguaggi XML sono domain specific, perché XML permette di creare facilmente nuovi linguaggi.

Un DSL può essere classificato come esterno o interno dipendentemente da come è designato e implementato: un DSL esterno è realizzato in modo da essere indipendente da un particolare linguaggio, uno interno, invece, è realizzato usando un linguaggio host ed è quindi limitato da questo per espressività e libertà di sintassi.

Se il linguaggio domain specific è esterno si possono definire i suoi elementi come la sintassi, i simboli e gli operatori attraverso la grammatica, ma occorre specificare il comportamento del compilatore per mappare e rendere eseguibile il codice realizzato. Questo problema non si ha con un linguaggio di tipo interno, ma il compromesso sta nella difficoltà di riuscire ad esprimere la sintassi che si ha in mente entro i vincoli posti dal linguaggio host. Inoltre occorre un error checking molto più esteso, perché spesso il codice è processato dinamicamente.

In questo testo l'attenzione viene posta sulla realizzazione di linguaggi di programmazione e DSL esterni. In particolare vengono presentati e confrontati tre strumenti per il loro sviluppo: JavaCC, ANTLR e Xtext. Al fine di poter comprendere più agevolmente alcuni concetti a cui si farà riferimento durante la rassegna di questi tools, viene ora discusso un capitolo di Nozioni Fondamentali.

2. Nozioni fondamentali

2.1 Lexical Analyzer e Parser

Il lexical analyzer (o lexer) è uno strumento che può leggere e spezzare una sequenza di caratteri in sottosequenze dette “token” e classificare questi ultimi in base ad un tipo.

Data la definizione formale di una grammatica, il parser prende in ingresso i tokens generati dal lexical analyzer, li analizza e li elabora in base ai costrutti specificati nella grammatica stessa.

Il parser determina la struttura del programma e genera un output, ad esempio sotto forma di albero che potrà poi essere utilizzato da un compilatore allo scopo di analizzare e generare il codice, oppure di valore numerico o restituisce la sequenza iniziale modificata, etc.

Nel caso il parser non produca nulla in output, il comportamento realizzato è quello di check lessicale e sintattico di un testo o di un linguaggio.

Esempio nel linguaggio C:

```
int main()
{
    return 0;
}
```

Il lexical analyzer del C divide la sequenza in

“int”, “ ”, “main”, “(”, “)”,

“ ”, “{”, “\n”, “\t”, “return”

“ ”, “0”, “ ”, “;”, “\n”,

“}”, “\n”, “”

attribuendo un identificativo ad ogni token, ad es.:

```
KWINT, SPACE, ID, OPAR, CPAR,
SPACE, OBRACE, SPACE, SPACE, KWRETURN,
SPACE, OCTALCONST, SPACE, SEMICOLON, SPACE,
CBRACE, SPACE, EOF.
```

Questa sequenza di tokens è poi passata al parser, che la analizza.

Analyzer e parser hanno anche il compito di generare errori nel caso l'input non sia conforme con le regole lessicali o sintattiche del linguaggio: l'analyzer controlla che i caratteri (o i gruppi di caratteri) che incontra nella sequenza in analisi rappresentino tutti tokens validi (cioè definiti nella grammatica), il parser controlla che l'utilizzo di questi tokens sia fatto nel modo corretto (cioè che la sequenza di tokens che riceve, opportunamente suddivisa, rispetti le regole).

Es. Se, dalla definizione della grammatica risulta che i tokens accettabili sono solo numeri, spazi ed il segno "+", quando la sequenza fornita in input è data da:

a) “5 - 6” → l'analyzer non riconosce il simbolo "-" e genera un errore

b) “5 ++ 6” → l'analyzer riconosce ogni simbolo come corretto e passa la sequenza di tokens al parser, è compito di quest'ultimo, in base alle regole dettate nella grammatica, controllare se la sequenza di tokens che riceve ha un ordine corretto o meno.

Se il parser si aspetta di ricevere un numero dopo un più, ma ottiene di nuovo un più, allora smette di accettare tokens dall'analyzer e genera un errore.

2.2 Backus-Naur Form (BNF) e forma estesa (EBNF)

Una grammatica context-free è una grammatica in cui ogni regola di produzione è nella forma V → w, dove V è un singolo simbolo non terminale (cioè un simbolo che sarà rimpiazzato da un gruppo di simboli terminali secondo le regole di produzione) e w è una stringa di simboli terminali (cioè simboli elementari del linguaggio, definiti da una grammatica formale) e/o non terminali e può essere vuota. Una grammatica è considerata context-free quando le sue regole di produzione possono essere applicate indipendentemente dal contesto non terminale: V può essere sempre sostituito da w.

BNF è una delle due notazioni principali per le grammatiche context-free (l'altra è la Van Wijngaarden form) ed è spesso usata per descrivere la sintassi di linguaggi di programmazione, il formato di documenti, set di istruzioni e protocolli di comunicazione: ossia in tutti quei casi in cui occorre un'esatta descrizione del linguaggio. Esistono diverse varianti come la forma estesa (EBNF) e quella aumentata (ABNF).

BNF è un metalinguaggio (a dire il vero è uno dei più vecchi linguaggi legati ai computer ancora in uso) che specifica un set di regole nella forma:

<symbol> ::= __expression__

dove <symbol> è un simbolo non terminale (denotato da "< >"), ::= significa che <symbol> dovrà poi essere sostituito da __expression__, __expression__ consiste in una o più sequenze di simboli (separate da "|", che indica un choice point).

I simboli che non compaiono mai nella parte sinistra dell'uguale sono detti simboli terminali e sono denotati da " ".

EBNF estende BNF potenziando l'espressività del linguaggio con l'utilizzo di due espressioni regolari (REGEX):

[ ] (equivalente a ( )?): rende l'elemento al suo interno opzionale

{ } (equivalente a ( )*): indica che l'elemento al suo interno più comparire 0 o più volte

e apportando ulteriori modifiche, come l'abolizione della notazione "< >" per i simboli non terminali e la sostituzione di "::=" con "=".

2.3 Choice points e lookahead

Nello scrivere il parser bisogna porre attenzione alla presenza di choice points, rappresentabili con l'espressione "[ ]" (equivalente a "( )?"). Gli elementi all'interno delle parentesi sono opzionali, cioè possono trovarsi nella sequenza in ingresso o essere ignorati. La presenza di choice points e la loro posizione all'interno delle regole può influenzare fortemente il tempo di mappatura della sequenza nel token corretto, se presente.

Infatti, quando l'elemento tra parentesi coincide con quello correntemente in analisi, l'analyzer assume inizialmente che il contenuto tra [ ] sia parte della sequenza. Se poi questa non corrisponde a nessuna di quelle definite nelle regole (perché ad es. i caratteri successivi non sono quelli attesi dalla regola), allora, per prima cosa, l'analyzer considera errata la sua supposizione, fa il backtracking e prova a rifare il match ignorando quello tra [ ]. Se la regola fosse :

```
void Input(): {}
{
  "a" BC() "c"
}


void BC(): {}
{
  "b" ["c"]
}
```

con la sequenza in input “abc”, l'analyzer prima proverebbe a controllare la presenza di ["c"] (dopo quella di "a" e "b"), ma poi sarebbe costretto a fare il backtracking, dal momento che il secondo "c" atteso non viene trovato, perché la sequenza di input è terminata.

Un altro tipo di choice point è dato dall’utilizzo del simbolo “|”, che permette di scegliere tra più opzioni. In questo caso è il parser che, in base al token in arrivo, potrebbe non saper scegliere quale opzione sia quella giusta.

All'aumentare dei backtrackings le performance diminuiscono drasticamente, perciò è possibile istruire l’analyzer e il parser perché prendano una decisione (se ignorare o meno il contenuto tra [ ], o quale delle possibilità offerte da “|” sia quella da scegliere) in base ai tokens successivi e poi procedano senza più fare backtracking. Il processo di esplorare i tokens successivi della sequenza in input è detto lookahead.

Per ridurre i problemi legati ai choice points si hanno due alternative: semplificare la grammatica o inserire degli hints per aiutare l’analyzer e il parser nelle decisioni da prendere, se permesso dal tool utilizzato.

2.4 Abstract syntax tree (AST)

Un abstract syntax tree è la rappresentazione sotto forma di albero della struttura sintattica astratta (cioè della struttura descritta come tipo di dato, ciò che il dato rappresenta) di un codice sorgente scritto in un linguaggio di programmazione. Ogni nodo dell'albero denota un costrutto del codice sorgente e la sintassi è astratta nel senso che non rappresenta ogni dettaglio della sintassi reale (ad es. il costrutto if-condizione-then-else può essere rappresentato da un nodo con tre figli, uno per la condizione, uno per l'espressione del then e uno per quella dell'else).

Gli AST sono il prodotto dell'analisi sintattica (o parsing) del compilatore, sono perciò usati come rappresentazione intermedia del programma nelle diverse fasi della compilazione e hanno un forte impatto sull'output finale del processo di compilazione.

2.5 Parser Top-Down e Grammatiche LL(n)

Un parse tree è un albero che rappresenta la struttura sintattica di una stringa in accordo con le regole definite da una grammatica context-free. Differisce da un AST perché la sua struttura e i suoi elementi riflettono più concretamente la sintassi del linguaggio in input.

Il parsing top-down è la strategia in cui dalla radice del parse tree si scende fino alle foglie utilizzando le regole della grammatica. I parsers LL sono un esempio di parsers che fanno uso di una strategia top-down. Questo approccio permette di utilizzare grammatiche più generali e più facili da debuggare.

Inizialmente si ipotizza una struttura generale, rappresentata dal parse tree, e poi si analizza una relazione tra gli elementi controllando se è compatibile con quella rappresentata dall'albero; questo metodo è utilizzato sia nell'analisi di linguaggi naturali che di quelli relativi ai computers.

L’analyzer fa il parsing dell'input di un linguaggio di programmazione controllando il match dei simboli incontrati con le regole di produzione (definite usando forme BNF). Un parser LL applica le regole di produzione alla sequenza, esaminando prima il simbolo più a sinistra della regola e procedendo con l’esame della relativa regola di produzione, se si tratta di un simbolo non terminale, prima di esaminare il simbolo successivo. Esempio:

Date le regole

```
A → aBC
B → c | cd
C → df | eg
```

per prima cosa il parser cerca di fare il match A → aBC, quando incontra B (simbolo non terminale) procede con il match B → c | cd e, risolto quello, prosegue con C → df | eg: quindi i tokens sono consumati da sinistra verso destra.

Se una stringa prodotta da una qualsiasi regola di produzione non inizia allo stesso modo di un'altra stringa prodotta da una delle altre regole, allora il linguaggio è detto non ambiguo. Il parsing di un linguaggio non ambiguo può essere fatto da una grammatica LL(1), dove (1) significa che il parser legge un token alla volta.

Per i linguaggi che non possono essere considerati non ambigui, invece, occorre che il parser faccia un lookahead maggiore di 1 simbolo (es. LL(3)).

3. JavaCC

3.1 Introduzione

JavaCC, o Java Compiler Compiler, è un parser generator e un lexical analyzer generator di tipo top-down LL(k) scritto puramente in Java.

JavaCC riconosce i files nel formato ".jj" e, in base al loro contenuto, genera un lexical analyzer e/o un parser in Java puro. Una sua caratteristica è data dal fatto che utilizza un unico file per specificare sia le regole lessicali che quelle sintattiche: ciò permette di utilizzare le prime nella specifica della grammatica, rendendola così più leggibile e manutenibile.

In un file nel formato ".jj", quindi, si specificano gli elementi della grammatica che l'analyzer prodotto dovrà essere in grado di riconoscere (cioè quelli accettabili) e tutte le possibili combinazioni legittime di questi (che il parser dovrà controllare). Tutto ciò che non viene specificato nel file di definizione della grammatica sarà poi considerato come errore dallo strumento apposito (analyzer o parser) durante l'elaborazione dei dati, che fornirà, per quanto possibile, la locazione esatta dell'errore e una diagnosi esaustiva.

Per specificare il comportamento del parser, si definisce una apposita classe Java con lo stesso nome del file di definizione della grammatica che la contiene, in cui si istanzia il parser come oggetto della classe.

Il costruttore del parser si aspetta di ricevere un InputStream (come System.in) o un Reader, per passarlo all'analyzer. Infatti, quando si chiama il metodo Start() dell'oggetto parser, questo invoca l'analyzer che, dopo aver esaminato la stringa in input, restituisce al parser una sequenza di tokens. A mano a mano che la riceve, il parser cerca una sotto-sequenza che faccia match con le regole definite nella grammatica.

Nel caso una sequenza di tokens soddisfi contemporaneamente più regole di produzione, allora si considera prima la regola che ha il match di caratteri più lungo, poi, in caso di parità, quella dichiarata per prima.

Riassumendo:

InputStream/Reader → lexical analyzer → tokens → parser → ???

???: qualcosa rappresentabile in Java, è il motivo per cui la grammatica è stata creata. Può essere un AST, un valore numerico (ad es. se si vuole realizzare un calcolatore), un codice assembly, il testo iniziale modificato, etc..

JavaCC è stato largamente utilizzato, ad es. per la produzione di

- parser per RTF, Visual Basic, Python, Rational Rose mdl files, XML, XML DTDs, HTML, C, C++, Java, JavaScript, Oberon, SQL, VHDL, VRML, ASN1;

- headers di email;

- diversi linguaggi proprietari.

Inoltre trova utilizzo anche per la realizzazione di lettori di file di configurazione, calcolatori, etc..

3.2 Funzionamento

JavaCC analizza la grammatica definita nel file in formato ".jj" e produce una classe Java (compilabile dal compilatore Java) per componente, tra cui:

- classi di errore (sottoclassi di Throwable):

    a) TokenMgrError, estende da Error perché non ci si aspetta che capiti e, comunque, non sarebbe recoverable. Questa eccezione è lanciata dall'analyzer quando incontra caratteri che non ha potuto riconoscere;

    b) ParseException, estende da Exception perché può e dovrebbe essere gestita a livello di codice. Questa eccezione è lanciata dal parser quando incontra sequenze di tokens che non ha potuto riconoscere;

- classe Token: ha un campo intero "kind" per rappresentare il tipo di token e un campo stringa "image" che rappresenta la sequenza di caratteri descritta dal token;

- classe SimpleCharStream: adapter che trasmette i caratteri all'analyzer;

- interfaccia: definisce il numero di classi usate dall'analyzer e dal parser;

e ovviamente il lexical analyzer e il parser!

Per fornire un comportamento più attivo al parser (e cioè far sì che produca in output un risultato), e non solo di controllo della sequenza di tokens, con JavaCC basta fornire una implementazione del metodo Start() nella classe di specifica della grammatica. Durante l'implementazione, che descrive un funzionamento dettagliato dell'utilizzo della sequenza di tokens in ingresso, si possono usare tutti i metodi delle classi di libreria Java. È poi JavaCC ad integrare queste funzionalità con il codice da lui automaticamente generato ed inglobare il tutto nella classe Parser.

3.3 Dettagli e modi d’uso

Il programmatore definisce nella grammatica le regole per generare tokens a partire da una sequenza di caratteri (JavaCC riconosce e gestisce qualsiasi carattere Unicode), utilizzando REGEX ed EBNF, che permettono di fare un minor utilizzo della ricorsione e di scrivere regole più leggibili. Definisce inoltre le regole di produzione di ciò che il parser dovrà restituire. Queste regole sono annotate con frammenti di codice Java e sono dette semantic actions.

Il numero di keywords predefinite di JavaCC è ridotto:

```
EOF, IGNORE_CASE, JAVACODE, LOOKAHEAD, MORE,
PARSER_BEGIN, PARSER_END, SKIP,
SPECIAL_TOKEN, TOKEN, TOKEN_MGR_DECLS
```

e alcune di queste verranno presentate nel seguito.

JavaCC offre all’analyzer concetti come:

- TOKEN: elemento composto da un nome e dalla sequenza di caratteri che rappresenta. Se marcato con “#”, il token è locale all'analyzer e non viene trasmesso al parser, può essere usato per migliorare la leggibilità e diminuire gli errori in caso di modifiche, quando si hanno blocchi di codice ripetuti;

- MORE: l’analyzer continua ad analizzare la sequenza e torna il token attuale assieme al successivo, come suo prefisso, utile ad es. con le sequenze di escape;

- SKIP: marca i tokens che non devono essere trasmessi al parser;

- cambi di stato: permettono di scrivere specifiche più comprensibili e di ricevere messaggi di error e warning più precisi.

Alcuni special tokens possono essere definiti nelle specifiche lessicali, questi vengono ignorati durante il parsing, ma possono essere utili perché utilizzabili da altri tools; un esempio ricorrente è l'utilizzo di tags nei commenti.

I tokens possono essere degli oggetti regolari o speciali, ogni token ha un campo detto specialToken che punta al token di tipo special a lui immediatamente precedente (se presente, a null altrimenti). I token regolari sono collegati in lista ed è quindi possibile scrivere semantic actions attraversando la sequenza di tokens, così che vengano eseguite durante il parsing.

Se la valutazione delle espressioni è fatta durante il parsing, per l'utilizzo di cicli occorre tradurre il corpo del loop in un codice intermedio (es. un albero) durante il parsing e poi eseguire questo codice ad ogni ciclo: infatti, il token manager non può tornare indietro ad ogni loop per rifare il parsing e la valutazione del corpo del ciclo (per motivi di performance, ad es.).

JavaCC usa e permette di settare l'utilizzo di lookahead. Quando si specifica LOOKAHEAD (k) da riga di comando o all'inizio del file di grammatica (nella sezione opzioni), si indica al parser che k (intero positivo) è il numero di tokens da controllare prima di prendere decisioni nei choice points, il default è 1. Per questo motivo i parsers generati da JavaCC sono detti parser LL(k). La specifica LOOKAHEAD (k) può essere fatta anche in locale (scelta consigliata), cioè in un punto specifico della grammatica prima di un choice point, così che la maggior parte di questa rimanga del tipo LL(1) e il parser abbia performances migliori.

JAVACODE è un modo per scrivere codice di produzione in Java piuttosto che in EBNF, è utile quando è difficile scrivere una grammatica per un determinato scopo o quando serve riconoscere qualcosa che non è context-free. JavaCC lo considera però come una black box, che in qualche modo compie la sua parte di parsing. Può quindi diventare un problema quando il JAVACODE è associato ai choice points come una delle possibilità.

JavaCC non automatizza la produzione di AST ma si appoggia a JJTree per farlo. JJTree è il preprocessore che JavaCC usa per la build dell'albero: l'output di JJTree è usato da JavaCC per creare il parser. JJTree espone inoltre un'interfaccia utilizzabile anche da altri parsers scritti ad hoc e fornisce un framework basato sul design pattern Visitor, che permette di attraversare l'albero di parsing in memoria.

JavaCC usufruisce anche di JJDoc per convertire i files di specifica per il parser in documentazione per la grammatica. JJDoc può operare in diverse modalità producendo testo o HTML.

3.4 Esempio

Un esempio solitamente proposto con JavaCC è la realizzazione di una calcolatrice. Qui se ne fornisce una versione semplificata in cui gli unici caratteri ammessi sono numeri interi (cioè le dieci cifre tra loro combinate), spazi e ritorni a capo (non ammessi tra le cifre di uno stesso numero) ed il segno "+". Utilizzando i costrutti già presentati, come SKIP, TOKEN, etc., è possibile definire il comportamento di analyzer e parser in un file formato ".jj" (in questo caso Adder.jj):

```
options {
    STATIC = false;
}
```

Nella sezione "opzioni" si trova un riferimento a STATIC. Quando questa opzione è settata a true (di default), tutti i metodi e le variabili della classe sono specificati come static nel parser che verrà generato: questo ne migliora le performance ma non consente l'utilizzo di più oggetti parser in contemporanea (occorre perciò re-inizializzare il parser ad ogni nuovo utilizzo).

Settando l'opzione a false, è possibile istanziare il parser come un qualsiasi oggetto Java con l'operatore new (utilizzabile anche nel contesto di più threads concorrenti), come mostrato qui di seguito:

```
PARSER_BEGIN(Adder)
class Adder {
    static void main(String[] args) throws
        ParseException,TokenMgrError,
        NumberFormatException{

        Adder parser = new Adder(System.in);
        int val = parser.Start();
        System.out.println(val);
    }
}
PARSER_END(Adder)
```

Tra i costrutti PARSER_BEGIN(Adder) e PARSER_END(Adder) è specificato il comportamento del parser, definendo una nuova classe Java contente il metodo main. Al suo interno viene istanziato il parser e chiamato il suo metodo Start(), il cui corpo è successivamente definito. In questo esempio di base è permesso al main lanciare eccezioni (le cui classi saranno automaticamente generate da JavaCC), sarebbe meglio, però, utilizzare il costrutto try-catch per catturarle e gestirle all'interno del main stesso.

```
SKIP: {" "}
SKIP: {"\n" | "\r" | "\r\n"}
TOKEN: {< PLUS: "+" >}
TOKEN: {< NUMBER: (["0"-"9"])+ >}
```

Il comportamento dell'analyzer è definito subito dopo: è istruito per ignorare (SKIP) gli spazi vuoti (" ") e i ritorni a capo ("\n" | "\r" | "\r\n") e per riconoscere come token validi (TOKEN) solo il segno "+" (< PLUS : "+" >) e una combinazione di un qualsiasi numero di cifre (< NUMBER : (["0"-"9"])+ >).

La notazione usata è simile a quella EBNF precedentemente descritta: da notare l'uso di parentesi angolari per la definizione di simboli (come in BNF) e dell'espressione regolare "+" che obbliga ad avere almeno una cifra affinché il carattere sia riconosciuto come NUMBER.

Il compito dell'analyzer è quindi quello di riconoscere nella sequenza in input numeri, "+", spazi vuoti e ritorni a capo, di generare errori nel caso di caratteri diversi da questi e di trasmettere al parser due soli tipi di token: PLUS e NUMBER.

```
int Start() throws NumberFormatException:{
    int val;  //valore del numero letto
    int sum;
}
{
    sum = Primary()

    (<PLUS> val = Primary() {sum += val;})*

    <EOF> {return sum;}
}

int Primary() throws NumberFormatException:{
    Token t;
}
{
    t = <NUMBER>;
    {return Integer.parseInt(t.image);}
}
```

Al parser vengono assegnati due compiti: controllare se la sequenza di tokens in ingresso rispetta un determinato ordinamento (<NUMBER> (<PLUS> <NUMBER>)* <EOF>, dove EOF è automaticamente riconosciuto e trasmesso dall'analyzer senza doverlo specificare) ed eseguire la sommatoria dei numeri ricevuti.

La prima sezione di ogni metodo, identificata da "{ }", è riservata alla definizione di variabili. L'utilizzo di metodi accessori (come Primary()) è opzionale, ma suggerito per evitare codice ripetuto. Nel definire il funzionamento del parser, si indicano in ordine i TOKEN attesi e si specificano le eventuali azioni da svolgere in relazione al TOKEN ricevuto.

In questo esempio il comportamento del parser è semplice: nel caso di <NUMBER>, memorizza il valore intero corrispondente (ricavato tramite la funzione di libreria di Java Integer.parseInt(String s)), nel caso di <PLUS> <NUMBER>, ricava il valore del secondo intero e lo somma al primo, nel caso la sequenza ricevuta non sia quella attesa, lancia una ParseException.

Per catturare la stringa corrispondente ad un token basta inizializzare una variabile di tipo Token (Token t = <NUMBER>;) e poi leggerne il campo image (t.image).

Esempio di sequenze in "input " → output:

- "15 + 4" → 19
- "15+4 " → 19
  "15
  +
  4" → 19

- "1.5 + 2.3" → ParseException perché l'analyzer non riesce a fare il match tra la stringa "1.5" e un token valido

- "1 5 + 4" → TokenMgrError perché il parser riceve la sequenza <NUMBER> <NUMBER> <PLUS> <NUMBER> <EOF> (non riconosciuta tra quelle specificate nelle regole)
- "15 + " → TokenMgrError perché il parser riceve la sequenza <NUMBER> <PLUS> <EOF> (come sopra)

4. ANTLR

4.1 Introduzione

ANTLR (ANother Tool for Language Recognition, pronunciato Antler) è un generatore di parser che può costruire e navigare parse trees, oltre che leggere, processare, eseguire e tradurre testi strutturati e files binari.

ANTLR, scritto in Java, permette di tradurre una grammatica in un parser/lexer per un linguaggio target (Java, C#, Python o JavaScript). Il target è specificabile tramite l'opzione -Dlanguage = <linguaggio target> da linea di comando, o installando ANTLR direttamente nell'ambiente di sviluppo (Eclipse, IntelliJ, Visual Studio, Maven) con un apposito plugin.

I file contenenti la grammatica sono specificati nel formato ".g4" e sono composti dal nome della grammatica, dalle sezioni dedicate (opzioni, import, token, canali, @azioni) e da un insieme di regole.

Per distinguere tra grammatiche per il parser e grammatiche per il lexer si possono usare due notazioni differenti nella specifica del nome della grammatica (parser grammar Name; e lexer grammar Name;). In assenza di queste la grammatica definita è una grammatica combinata che può contenere regole per entrambi.

ANTLR permette di specificare il linguaggio tramite una grammatica context-free e usando notazioni EBNF. Il parser generator produce parsers adattivi (analizzano cioè la grammatica a runtime, cercando di fare ottimizzazioni) di tipo LL(k).

A differenza di altri tools di grammatica, ANTLR permette l'uso di regole ricorsive per la definizione del parser, semplificandone la scrittura grazie ad espressioni più vicine al linguaggio naturale:

```
Expr : expr '*' expr
     | expr '+' expr
     | INT;
```

Il tool ANTLR è stato utilizzato in diversi progetti, come:

- Groovy, Jython, Hibernate, Apex, IntelliJ IDEA, Clion e Apache Cassandra;

- OpenJDK Compiler Grammar project (versione sperimentale del compiler javac basata su una grammatica scritta in ANTLR);

- il linguaggio di query di Twitter;

- server Weblogic.

4.2 Funzionamento

Dopo l'analisi della grammatica presente nel file in formato ".g4", ANTLR genera le classi (nel linguaggio target, d'ora in avanti verrà considerato Java come linguaggio target) BaseListener.java, Listener.java, Lexer.java, Parser.java e i files Grammar.tokens GrammarLexer.tokens (dove Grammar è il nome scelto per la grammatica).

Per capire meglio come è strutturato un file di grammatica per ANTLR è comodo partire da un piccolo esempio, che riprende quello visto in JavaCC.

Il lexer è istruito per riconoscere numeri (composti da 1 o più cifre, grazie alla REGEX "+") e per ignorare spazi vuoti e ritorni a capo ("→ skip"). Il parser riconosce sequenze di interi separati da "," e ne fa la somma stampando a schermo il risultato ottenuto.

```
grammar Adder;

@header{
    package foo;
}
```

```
@members{
    int count = 0;
}

list
@after {System.out.println(count+" ints");}:
    INT {count++;} (',' INT {count++;})*;

INT: [0-9]+ ;
WS: [\r\t\n]+ -> skip;
```

Questo è un esempio di grammatica combinata: a parte una sezione iniziale di dichiarazione delle azioni, sono specificati sia il comportamento del parser che del lexer. Le azioni, identificate dal carattere "@", sono definite dal linguaggio e permettono di realizzare uno specifico comportamento interagendo con le classi generate da ANTLR. In particolare, @header{ } è utilizzata per iniettare codice prima della definizione della classe (ad es. per la specifica del package) e @members{ } per iniettarlo tra le definizioni, i campi e i metodi della classe generata (ad es. per la dichiarazione di variabili). @after{ }, infine, permette di eseguire azioni dopo che il parser ha esaurito il suo compito, in questo caso per stampare a schermo il risultato del calcolo.

Blocchi di codice, scritti nel linguaggio target e racchiusi tra "{ }", possono essere inseriti nella grammatica per fornire un determinato comportamento a lexer e parser (es. {count++;}). Da ANTLR3 ad ANTLR4, però, è stato introdotto un nuovo meccanismo che permette di definire il comportamento attivo del parser in linguaggio target in un file apposito (estendendo ad es. la classe BaseListener), così che la grammatica definita sia il più possibile indipendente dal linguaggio target e più facilmente riutilizzabile.

Dopo che il lexer ha suddiviso la stringa di input in tokens, il parser crea il parse tree, in cui le foglie sono costituite dai tokens in input e i nodi interni corrispondono a nomi che raggruppano e identificano logicamente i figli (nodename). Se si applica una label (costrutto #label) ad ogni alternativa di una regola che presenta choice points, il parse tree conterrà un nodo per ogni choice (identificato dalla #label applicata).

ANTLR genera automaticamente il parse tree e gestisce un ParseTreeWalker in grado di navigare l'albero e comunicare eventi ai listeners. Inoltre ANTLR predispone delle interfacce per creare listeners o usare visitors; la differenza più importante tra i due è che i metodi del listener sono chiamati automaticamente dall'oggetto walker che visita l'albero, mentre quelli del visitor devono essere chiamati esplicitamente per visitare i nodi figli (se si omette di chiamare i metodi del visitor sui figli di uno nodo, il suo sottoalbero non viene esaminato).

ANTLR mette a disposizione anche un meccanismo di tree pattern, usato per testare se uno specifico sottoalbero ha una determinata struttura e per estrarre i discendenti del sottoalbero, in base alla struttura stessa. L'idea principale è che l'analisi viene fatta ad un livello di sottoalbero piuttosto che di nodo. Il tool converte l'espressione da controllare "<ID> = <expr>" in un parse tree che ha la particolarità di avere i nodi che rappresentano ogni sottoalbero con token "ID" e regola "expr". Viene poi chiamato il metodo compileParseTreePattern del parser, che necessita di sapere a quale regola della grammatica fa riferimento il pattern cercato. A questo punto si fa il match ed eventualmente si ricavano dal parse tree originario i tokens corrispondenti (se più di uno).

Un ulteriore strumento messo a disposizione per l'identificazione di (set di) nodi in un parse tree è XPath. I paths di XPath sono stringhe che rappresentano i nodi o i sottoalberi che si vogliono selezionare in un parse tree, sono costruite con una serie di nodenames separati da simboli:

"/": indica che il prossimo elemento è un root node;

"//": indica tutti i nodi dell'albero che fanno match con il prossimo elemento del path;

"!": indica tutti i nodi dell'albero eccetto quelli relativi al prossimo elemento del path.

Tra i meccanismi presentati in sostituzione al codice target inline alla grammatica è preferibile utilizzare :

XPath → quando si cercano nodi specifici in un certo contesto;

tree pattern → quando si vogliono trovare degli specifici sottoalberi;

interfacce listener/visitor → quando si vogliono visitare molti (o tutti i) nodi di un albero, perché permettono di fare calcoli nel linguaggio target e salvare le informazioni necessarie.

4.3 Dettagli e modi d'uso

È possibile importare grammatiche, ereditando le regole, i tokens e le @azioni, ma la sezione opzioni delle grammatiche importate viene ignorata. Le regole nella nuova grammatica fanno l'override di quelle della grammatica importata implementando il concetto di inheritance (similmente al modello object-oriented). Quindi l'import non è una semplice inclusione di codice, ma il tool ANTLR carica tutte le grammatiche importate in oggetti grammatica e poi fa il merge di regole, tokens e azioni con quelle della grammatica corrente.

Ovviamente ci sono delle limitazioni sul tipo di grammatiche che si possono importare: le grammatiche specifiche possono importare solo altre dello stesso tipo, mentre quelle combinate possono importarle tutte.

Le keywords di ANTLR sono limitate e, in parte, ricalcano quelle di Java:

import, fragment, lexer, parser, grammar, returns, locals, throws, catch, finally, mode, options, tokens.

ANTLR fornisce inoltre una serie comandi per il lexer, indipendenti dalla linguaggio target, che vanno inseriti dopo l'ultima alternativa (quella più esterna in caso di choice points) con la sintassi "->":

- skip: il token corrente non viene trasmesso al parser;

- more: il lexer viene forzato a ricavare il token successivo (il type del token diverrà quello dell'ultimo considerato);

- mode(x) : altera lo stack mode, cambiando la mode corrente. Con i comandi popMode e pushMode(x) si può navigare lo stack mode;

- type(x): setta il type associato al token;

- channel(x): setta il numero di canale riferito al token.

I tokens supportano alcuni attributi predefiniti e read-only, necessari per leggerne le proprietà come il tipo e il testo in input che ha fatto match. Con la sintassi $token.attribute le azioni possono accedere alle varie proprietà del token, dove token può essere o il nome stesso del token (rappresenta l'istanza, se il nome del token è presente una sola volta) o il nome del riferimento all'istanza (se il nome del token compare più volte: a=INT, poi viene referenziato con $a.attribute, es. $a.line).

I diversi attributi leggibili dall'istanza sono:

-text: torna una stringa contenente il testo che ha fatto match con la specifica del token;

-type: l'intero associato al tipo del token;

-line: il numero di riga (a partire da 1) in cui compare il token nella sequenza di input;

-pos: la posizione all'interno della linea (a partire da 0);

-index: l'indice che identifica il token tra tutti quelli trasmessi dal lexer;

-channel: il numero di canale riferito al token (0 di default);

-int: il valore numerico relativo al testo del token, assumendo che questo sia una stringa numerica valida (corrisponde al metodo Java Integer.valueOf ($token.text)).

Anche le regole del parser hanno degli attributi (la sintassi per accedervi è la stessa di quella per i tokens), tra cui:

-start: referenzia il primo token che potenzialmente fa il match con la regola, se una regola non fa il match con nessun token questo attributo punta al primo token che avrebbe potuto fare il match;

-stop: referenzia l'ultimo token non nascosto che fa il match con la regola. Questo attributo è utilizzabile sono nelle azioni @after e @finally, se riferito alla regola corrente.

Una caratteristica propria di ANTLR è l'utilizzo del dynamic scoping: in una catena di chiamate a metodi, quelli più interni possono accedere a variabili locali definite in quelli più esterni. Non è quindi possibile decidere a compile time il set di variabili visibili da ciascun metodo, perché questo set dipende da chi invoca il metodo. ANTLR fa uso del dynamic scoping permettendo alle azioni di accedere agli attributi delle regole che le invocano: ciò viene fatto con la sintassi $r::x, dove r è il nome della regola e x ne rappresenta un attributo.

Nel lexer si possono definire fragments, che sono regole che non producono tokens visibili al parser, ma sono utili per rendere più leggibile il codice.

Es.

INT : DIGIT+ ; // fa riferimento alla regola DIGIT

fragment DIGIT : [0-9] ; // questa è una helper rule, non produrrà tokens

ANTLR mette a disposizione anche il concetto di mode all'interno di una lexer grammar: si possono raggruppare le regole in base al contesto (se non specificato, tutte le regole sono nella default mode). Il lexer potrà poi produrre tokens che fanno match con le regole nella mode corrente (se non specificato diversamente, la mode corrente è la default mode).

Il lexer mette a disposizione i costrutti ".." (detto range operator) e "[ ]" (notazione character set) per semplificare la scrittura di regole: il primo equivale alla notazione "-" (es. a..z è equivalente ad a-z), mentre il secondo è tipico delle REGEX.

Per la scrittura di choice points, uno strumento utile è rappresentato dai semantic predicates, posti alla sinistra dell'alternativa: consiste in un predicato p (scritto nel linguaggio target) identificato con la notazione {p}?. La valutazione di p viene poi fatta a runtime e se p risulta essere false, l'alternativa corrispondente nella regola viene ignorata. Un predicato si dice visibile se viene controllato prima di una azione o di un token nella fase di predizione, solo i predicati visibili vengono valutati, gli altri sono ignorati per motivi di semantica e coerenza. Una azione non può essere eseguita prima che la scelta sia stata effettuata (per motivi di non reversibilità di alcune azioni), perciò, in presenza di azioni e predicati, se l'azione si trova prima del predicato, questo viene ignorato perché il suo risultato potrebbe dipendere dall'effetto dell'azione.

È possibile l'uso di predicati che dipendono da parametri o variabili locali alla regola, ma in certi casi questi vengono ignorati perché non possono essere valutati nel contesto corretto: con più chiamate consecutive alla stessa regola, ad esempio, ad ogni chiamata la variabile locale è fornita in una nuova copia re-inizializzata e perciò non influenzata da eventuali modifiche fatte alle chiamate precedenti. Esempio:

```
prog: stat+;
stat locals [int i=0]: {$i==0}?
    'if' expr 'then' stat {$i=5;}
    ('else' stat)? | 'break' ';'
```

In questo esempio il predicato "{$i==0}?" sulla variabile locale "i" viene ignorato perché, se si trovano più "stat" consecutivi (permesso da "prog: stat+"), ognuno avrà la sua copia di "i".

I predicati sono disponibili anche nelle regole del lexer e possono essere utili per scegliere quale regola seguire in caso di più matches con la stessa sequenza in input: quando il lexer incontra un predicato che risulta essere falso, l'intera regola viene ignorata. Anche qui le azioni del lexer devono comparire dopo i predicati (perché non possono essere eseguite prima di aver valutato il predicato e perché questo non può dipendere dal loro effetto).

Un esempio banale può essere il seguente, in cui le due regole del lexer sono equivalenti, anche se la seconda è ovviamente molto più efficiente:

ENUM: [a-z]+ {getText().equals("enum")}?

ENUM : 'enum' ;

Tutte le sottoregole EBNF come (...)?, (...)* e (...)+, sono dette "greedy" perché cercano più match possibili nella sequenza in input. Alcune volte ciò non è desiderabile e perciò ANTLR adotta un "?" addizionale dopo tali notazioni per indicare che le sottoregole così ottenute sono "nongreedy" e agiscono diversamente: consumano il minore numero di caratteri possibile per soddisfare la regola.

Esempio di commento C-like:

COMMENT: '/*' .*? '*/' -> skip ; // .*? fa il match con qualunque carattere fino al primo "*/"

Un fattore a cui bisogna però prestare attenzione è la presenza di regole nongreedy combinata con l'uso di choice points: se ad esempio la regola fosse FIRSTLETT: (’a’|’ab’)*?; , indipendentemente dall'input, la seconda choice sarebbe unreachable perché la regola nongreedy si ferma alla più piccola sequenza che fa match. Quindi se anche la sequenza in input fosse "ab", la regola troverebbe il match con il carattere "a", fermandosi.

Anche nel parser si può fare uso di sottoregole nongreedy per far sì che il match venga fatto con la più corta sequenza di tokens che soddisfi la regole di parsing.

Infine è implementato un meccanismo per cui se in una regola è lasciata un'alternativa vuota, allora significa che l'intera regola è opzionale (es. superClass: 'extends' ID | ;).

4.4 Esempio

Per avere un riferimento più nel dettaglio per quando riguarda il funzionamento di ANTLR4, di seguito viene presentata la grammatica relativa ad un semplice calcolatore.

In un unico file di grammatica vengono specificati i compiti di lexer e parser, il comportamento da assumere in base ai tokens ricevuti dal parser verrà definito poi nel linguaggio target (Java in questo caso) tramite l'utilizzo di pattern (visitor o observer).

Da notare che nella versione 4, l'unico codice in linguaggio target che compare nella grammatica è "@header{package calculator;}" (non compaiono infatti altre actions), questo per renderla il più possibile riutilizzabile.

```
grammar Calculator;


@header {
    package calculator;
}
```

```
// PARSER
Program: (assignment ';')+;
assignment: ID '=' expression;
expression: '(' expression ')'
                          #parentExpression
    | expression ('*'|'/') expression
                          # mulOrDiv
    | expression ('+'|'-') expression
                          # addOrSubtract
    | INT
                          # integer;
```

Le diverse alternative della regola "expression" sono marcate con una label per far sì che nel parse tree generato corrispondano a nodi distinti. Inoltre, l'ordine con cui sono dichiarate le diverse alternative genera una precedenza nella valutazione delle espressioni. Come evidenziato dalla regola "expression", ANTLR permette l'uso di left recursion (perché traduce in modo trasparente tutte le regole ricorsive in regole non ricorsive); non permette però indirect left recursion (cioè una regola x che fa riferimento ad una regola y che fa a sua volta riferimento ad x).

```
// LEXER
ID: ('a'..'z'|'A'..'Z')+;
INT: '0'..'9'+;
WS: [\t\n\r]+ -> skip ;
```

Il lexer definisce gli elementi della grammatica (tokens) usando solo lettere maiuscole, in particolare sono ammessi solo nomi (ID), che contengono le lettere dell'alfabeto, numeri interi e spazi/ritorni a capo, che non verranno poi trasmessi al parser ("->skip").

Per utilizzare poi le classi prodotte da ANTLR, si crea una classe Calculator nel package indicato nella grammatica.

```
package calculator;

import org.antlr.v4.runtime.ANTLRFileStream;
import
    org.antlr.v4.runtime.CommonTokenStream;
import
    org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.Token;


public class Calculator{
  public static void main(String[] args)
                            throws Exception{
(1) CalculatorLexer lexer = new
                CalculatorLexer( new
                ANTLRFileStream(args[0]));
(2) CommonTokenStream tokens = new
                CommonTokenStream(lexer);
(3) CalculatorParser p = new
                CalculatorParser(tokens);
(4) p.setBuildParseTree(true);
(5) p.addParseListener(new
                CalculatorListener());
(6) ParserRuleContext<Token> t = p.program();
  }
}
```

Si crea il lexer (1) e lo si usa per generare i tokens (2), si crea poi il parser passandogli la sequenza di tokens (3). Il parser viene poi istruito per generare il parse tree (4). Viene indicato il listener al parser (5) per poi chiamare il programma (6).

Il listener CalculatorListener estende l'interfaccia fornita da ANTLR CalculatorBaseListener (che fornisce metodi per modellare il funzionamento del parser a seconda della regola) e realizza il comportamento del calcolatore eseguendo le operazioni necessarie. In questo caso i dati possono essere manipolati, ad esempio, gestendo uno stack di interi:

```
//All'interno della classe CalculatorListener
public Stack<Integer> stack = new
                          Stack<Integer>();


...


public void
exitAddOrSubtract(AddOrSubtractContext ctx){
    Integer op1 = stack.pop();
    Integer op2 = stack.pop();
    if (ctx.getChild(1).getText().
                              equals("-")){
      stack.push(op2 - op1);
    } else {
      stack.push(op1 + op2);
    }
}
```

Gli oggetti utilizzati (come ctx) appartengono a classi (AddOrSubtractContext) generate automaticamente da ANTLR in base alle regole definite nel parser.

Esempio di sequenze in "input " → output:

- "a = 1+2;" → a = 3
- "b = 2*3;" → b = 6
- "c = (1+2)*3" → c = 9
- "d = 2+2*2" → d = 6

- "a1 = 1+2;" → Exception perché il lexer non riconosce "a1" come ID valido
- "b = 2**3;" → Exception perché il parser non riconosce INT "*" "*" INT come sequenza di tokens valida

5. Xtext

5.1 Introduzione

Xtext è un framework per lo sviluppo di linguaggi di programmazione, che fornisce come supporto anche le funzionalità dell'ambiente Eclipse.

Dopo aver specificato la grammatica del linguaggio, Xtext genera un'implementazione, il cui compilatore e i suoi componenti (parser, AST type-safe, serializer, formattatore di codice, framework per lo scoping, generatore di codice/interprete, etc.) sono indipendenti da Eclipse e possono essere usati in ambiente Java. È possibile specificare il comportamento di componenti runtime integrabili con il framework di Eclipse, che mette già a disposizione una IDE configurabile specifica per il linguaggio. Nel seguito verrà presentato l'utilizzo di Xtext associato al framework Eclipse.

Per istruire Xtext sulla grammatica da utilizzare, viene creato un nuovo progetto Xtext in Eclipse, che a sua volta genera 4 progetti: quello in cui viene specificata la grammatica e i componenti runtime, più tre di supporto e configurazione (sdk, tests e UI). Una volta scritta la grammatica, occorre generare l'infrastruttura del linguaggio eseguendo il code generator, che deriva i vari componenti tramite il file in formato ".mwe2", automaticamente generato. Infine, lanciando dal progetto principale una nuova istanza di Eclipse, è possibile utilizzare l'editor per testare il corretto comportamento della grammatica realizzata: ogni nuovo file in formato ".<estensione scelta per i file del DSL>" viene riconosciuto da Eclipse come file contenente codice scritto nel nuovo linguaggio.

Il framework fornisce supporto alla scrittura di codice nel linguaggio appena definito, come la funzionalità del content assist, che suggerisce un input valido per la posizione corrente nel file, oltre a quelle di highlight e coloring della sintassi e di validazione e linking che verranno riprese in seguito.

Prima di entrare più nel dettaglio, è necessario introdurre brevemente due linguaggi su cui Xtext si appoggia: Xbase e Xtend.

Xbase è un linguaggio di programmazione parziale, implementato in Xtext, che può essere utilizzato inline con quest'ultimo. Ha una sintassi simile a Java (a parte alcune differenze, di cui la principale è che tutto è un'espressione, non ci sono statements), e mette a disposizione l'inferenza sul tipo, le lambda expressions ed altri meccanismi. Ha associati un interprete e un compilatore che producono codice Java per facilitare l'aggiunta di caratteristiche al DSL e per renderlo eseguibile.

È stato realizzato con l'idea di semplificare la costruzione di espressioni (che definiscono il comportamento del linguaggio di programmazione) fornendo anche implementazioni di default per aspetti come content assist, syntax coloring e navigazione.

Xtend è un linguaggio di programmazione che usa tipi di dato statici e che può essere tradotto in codice Java. Riprende da Java diversi aspetti, integrandolo con nuove funzionalità grazie ad una libreria di utilities ed estensioni posta logicamente sopra al JDK. Così come in Xbase, anche in Xtend tutto è un'espressione ed è possibile fare uso di proprietà per accedere ai campi di un oggetto.

```
val it = new Person
name = 'John'
        //corrisponde a it.setName("John");
if (it.name == "Bob"){
        //tradotto in it.getName()
```

Grazie ad Xtend si può fare l'override di diverse funzionalità automaticamente offerte da Xtext. Ad es. modificando il code generator (file Generator.xtend) si può iterare tra tutti gli elementi del source file (grazie ad un TreeIterator) filtrandoli per tipo e realizzando un certo comportamento (es. può essere generata una apposita classe java per ogni elemento individuato, il cui contenuto è dichiarabile direttamente in Xtend).

Xtext è utilizzato nel campo dell'automazione, dei dispositivi mobili, dei sistemi embedded e nello sviluppo di giochi. I linguaggi costruiti con Xtext sono utilizzati per progetti open-source come Maven, Eclipse B3, la piattaforma Eclipse Webtools e Google's Protocol Buffers, oltre che nel campo della ricerca.

5.2 Funzionamento

Ogni file di grammatica inizia con un header che specifica alcune proprietà della grammatica stessa: il nome del linguaggio che si vuole realizzare e la dichiarazione della super grammatica da utilizzare (meccanismo detto grammar mixin), che di default è org.eclipse.xtext.common.Terminals, che offre un set di regole terminali già implementate.

Se la grammatica in costruzione eredita dalla super grammatica org.eclipse.xtext.xbase.Xbase, invece che da Terminals, si possono utilizzare espressioni e tipi definiti nel linguaggio Java assieme al codice del DSL: es. XImportSection può essere utilizzato per specificare una sezione di Import che funziona esattamente come quella Java, oppure si può fare uso di tipi di dati Java (come String), dei generici e delle espressioni (come le lambda expressions).

Per ottenere ciò, occorre definire nella grammatica espressioni dipendenti da concetti Java

Es. ... 'extends' superType=JvmTypeReference ...

//JvmTypeReference definisce la sintassi per i nomi Java

e specificare il comportamento della classe JvmModelInferrer.xtend. Questa classe è automaticamente generata ed è utilizzata per mappare i concetti del DSL in quelli di altri linguaggi (in questo caso Java), così che Xtext sappia come interpretarli ed eseguirli.

In generale, il model inferrer è una API che permette di mappare concetti ed espressioni, definendone anche lo scope. Così facendo, quindi, è possibile utilizzare in contemporanea codice del DSL e di Java nella stessa applicazione.

Nel definire il proprio DSL è possibile inserire dei controlli statici per i vincoli (es. lanciare un warning se l'identificatore di una regola non inizia con la maiuscola, o un errore per dichiarazioni differenti con lo stesso identificatore): basta esprimere il vincolo sotto forma di metodo (marcato con l'annotazione "@Check") nella classe Validator e il framework validerà i controlli in automatico.

Xtext genera inoltre un progetto per la gestione di test con Junit4, in cui, grazie a classi Xtend e metodi di assert, è possibile testare il funzionamento di parser e linker.

I parsers di Xtext creano grafi in memoria (semantic models) mentre leggono ed elaborano la stringa in input; per fare ciò Xtext si appoggia internamente ad EMF (Eclipse Modeling Framework). I modelli di EMF sono composti da istanze di EObjects (oggetti di Ecore) connesse tra loro e rappresentano in memoria, sotto forma di AST, il testo di cui è stato fatto il parsing. Il modello prodotto dall'utente genera quindi un metamodello (Ecore) che a sua volta istanzia Ecore.

Riassumendo:

Modello utente → modello Ecore (metamodello) → istanza Ecore

EMF ha associato anche una code generator che genera classi Java dal modello Ecore, aggiungendo informazioni.

Di default Xtext inferisce il modello Ecore direttamente dalla grammatica tramite il comando "generate name namespaceURI". È possibile importare EPackages già esistenti usando il loro URI e cambiando delle configurazioni nel workflow. Nel caso di più imports è possibile utilizzare un alias per identificare l'EPackage specifico e poi riferirsi ai suoi elementi con la sintassi alias::Elem.

È permesso combinare l'utilizzo di import e di generate (diversi modelli Ecore vengono importati, viene assegnato loro un alias e viene prodotto un unico modello Ecore finale), ma è sconsigliato perché può portare a problematiche difficili da inquadrare.

5.3 Dettagli e modi d'uso

Xtext si appoggia sul grammar language, un DSL realizzato per la descrizione di linguaggi testuali: il parser produce un semantic model a mano a mano che consuma il file di input.

In Xtext il termine "parsing" comprende diversi meccanismi, ognuno specificabile con regole apposite, che sono riassumibili in lexing, parsing, linking e validation. Nel seguito verranno presentate alcune caratteristiche proprie di ciascuna fase.

Nella prima fase, il lexing, una sequenza di caratteri in input è trasformata in una sequenza di tokens (simboli atomici fortemente tipizzati, identificati da una regola terminale o da una keyword) utilizzando espressioni EBNF combinate a sequenza di escape e caratteri Unicode.

Es. terminal ID : ('^')?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

Questa terminal rule, o token/lexer rule, è definita nella grammatica Terminals (che di solito viene ereditata dalla grammatica che si vuole realizzare con la clausola "with") e stabilisce che ID è un token che identifica una sequenza che inizia opzionalmente con "^" (si noti il "?"), seguito da una lettera o un underscore e da un certo numero di lettere, underscores, o cifre.

Il simbolo "^", detto caret, è utilizzato per fare l'escape di un identificatore nel caso di conflitti con keywords già esistenti ed è automaticamente rimosso.

Più in generale, le regole terminali sono da definire come segue:

```
TerminalRule:
    'terminal' name=ID ('returns'
                              type=TypeRef)?
    ':' alternatives=TerminalAlternatives
    ';'
;
```

Ogni regola terminale ritorna un valore atomico, EString di default, specificabile con la clausola 'returns':

```
terminal INT returns ecore::EInt : ('0'..'9')+;
```

ad es. è la definizione della regola terminale che identifica gli interi.

In particolare, è possibile far tornare alla regola un qualsiasi tipo di dato che sia istanza di ecore::EDataType, occorre poi istruire Xtext su come convertire la stringa di testo letta nel tipo di dato scelto, implementando il servizio IValueConverterService.

È importante notare che l'ordine con cui vengono dichiarate le regole terminali è fondamentale, perché, in caso di conflitti, quelle dichiarate per ultime prevalgono su quelle dichiarate per prime (ciò vale anche in caso di conflitti con regole di grammatiche importate: queste ultime vengono ignorate).

Tra le espressioni EBNF, si possono utilizzare gli operatori di cardinalità (?, *, + o nessun operatore per indicare esattamente una corrispondenza), il range operator ".." precedentemente incontrato e la wildcard "." per indicare un qualsiasi carattere. Un altro meccanismo utile è il così detto until token "->" che funge da wildcard includendo qualsiasi carattere fino ad una certa occorrenza specificata:

terminal ML_COMMENT : '/*' -> '*/';

nel commento mutlilinea qualsiasi carattere compreso tra "/*" e "*/" viene riconosciuto come parte del token ML_COMMENT grazie alla presenza dell'until token.

Infine, si può fare riferimento a regole già definite con il meccanismo di rule calls ed è possibile utilizzare dei terminal fragments per rappresentare sequenze di caratteri che verranno poi utilizzate più volte nelle regole terminali.

Le parser rules non producono singoli token terminali ma un albero di token terminali e non, detto anche parse tree o node model in Xtext. Queste regole entrano in gioco nella creazione degli EObjects che formano il semantic model (AST) e perciò sono chiamate anche regole di produzione o EObject rules.

Alcune funzionalità già incontrate (come range operator, wildcard e until token) non sono disponibili per la scrittura di parser rules, mentre restano validi i gruppi (elementi raggruppati tra parentesi), le alternative (separate da "|"), le keywords e le rule calls. Esistono anche meccanismi usati per definire come l'AST debba essere costruito e alcuni sono elencati di seguito.

Se una parser rule non specifica il tipo dell'oggetto di ritorno (la sua EClass), allora il nome del tipo corrisponde al nome della regola. Le componenti di una regola del parser sono assegnate a dei campi della regola stessa: se l'assegnamento è di un singolo valore si usa la notazione "campo = valore" (dove valore può essere un ID, una keyword, una cross reference), altrimenti si possono usare le notazioni "+=" (per indicare che si ha più di un valore, i.e. una lista) o "?=", operatore booleano (se il valore è presente, il campo vale true, false altrimenti). Inoltre si può fare uso delle cardinalità nella scrittura delle regole (?, * e +). Le keywords del linguaggio che si vuole realizzare sono definite usando delle stringhe.

Una particolarità di Xtext è la possibilità di utilizzare cross-references, ossia informazioni cross-link specificate nella grammatica e risolte durante il linking.

Il termine che funge da cross-ref è inserito tra "[ ]" e fa riferimento alla EClass restituita da una determinata regola, non alla regola stessa!

Il simbolo "|" nel contesto di una cross ref non indica un'alternativa, ma specifica la sintassi della stringa di cui è stato fatto il parsing. Esempio:

```
Entity:
  'entity' name = ID
  ('extends' superType = [Entity |
                    QualifiedName])?
  '{'
    (features += Feature)*
  '}'
;


QualifiedName:
  ID ('.' ID)*
;
```

La riga "('extends' superType = [Entity | QualifiedName])?" indica che la clausola opzionale "extends" (si noti il "?" a fine blocco) è seguita da una entità, il cui valore è assegnato al campo superType e la cui sintassi è data da QualifiedName.

È possibile fare uso di unordered groups (utili ad esempio nella definizione di proprietà come static, final e della visibilità): sono elementi, anche opzionali, separati da "&" che possono comparire in qualsiasi ordine, ma una sola volta. Se l'elemento fa riferimento a più occorrenze di una certa stringa si possono utilizzare gli operatori "+" e "*", ma questo elemento può essere scritto una sola volta nella definizione della regola:

Property: (static ?= 'static')? & (final ?='final') ? & vals*=INT;

In questo modo le keywords “static” e “final” sono opzionali e possono comparire in qualsiasi posizione, inoltre possono essere presenti più numeri interi consecutivi (vals è scritto una sola volta nella definizione della regola).

Solitamente l'oggetto da ritornare dalla regola è creato all'occorrenza del primo assignment: creazione ritardata (o lazy). Con le Actions è però possibile istruire Xtext affinché la creazione dell'EObject sia fatta esplicitamente. Esempio:

```
MyRule returns TypeA:
  "A" name=ID |
  "B" {TypeB} name=ID
;
```

La notazione “{TypeB}” fa sì che venga creata un'istanza di TypeB e che questa venga assegnata al risultato della regola. Questa notazione semplifica anche la scrittura delle regole, evitando l’introduzione di regole delegate per creare nuovi oggetti:

```
MyRule returns TypeA :
  "A" name=ID |MyOtherRule
;

MyOtherRule returns TypeB :
  "B" name = ID
;
```

infatti, questa notazione è equivalente alla precedente, ma meno concisa.

Utilizzando parsers LL, non è possibile scrivere espressioni left recursive, perciò Xtext mette a disposizione una funzionalità detta Assigned Actions che permette, tramite comandi che manipolano l'albero in costruzione, di realizzare lo stesso comportamento che si otterrebbe facendo uso di left recursion.

Inoltre è possibile l'uso di hidden terminal symbols, cioè di caratteri che vengono inseriti nel node model, ma che non sono rilevanti per il modello semantico (cioè non vengono considerati dal parser).

Il tipico esempio è quello di spazi bianchi e ritorni a capo: questi, come ogni altro simbolo che può diventare hidden, possono essere definiti tali sia a livello di intera grammatica che di singola parser rule.

È possibile definire regole per il parser che creano istanze di EDataType, invece che di EClass; queste regole sono dette datatype rules. Esempio:

```
QualifiedName returns ecore::EString :
  ID ('.' ID)*
;
```

Come per le terminal rules il tipo di ritorno può essere specificato (EString di default), si hanno inoltre i vantaggi dell'utilizzo di hidden tokens e del fatto che si è nel parser: la regola terminale ID non verrà nascosta dalla regola del parser QualifiedName. Se una parser rule non ne chiama altre e non contiene azioni o assignments allora è una datatype rule.

Xtext permette di abilitare il backtracking per il suo ANTLR parser generator, ma non è raccomandato perché influenza la correttezza dei messaggi di errore. In alternativa, è possibile fare uso di syntactic predicates per gestire i messaggi di errori di ANTLR durante la generazione del parser. I syntactic predicates sono identificati dal simbolo "=>" e sono utilizzati per indicare al parser cosa fare in determinate situazioni ambigue. Esempio:

```
if (isTrue())
  if (isTrueAsWell())
    doStuff();
  else
    doOtherStuff();
```

Nel caso di if annidati con un solo else, il parser ha bisogno di capire a quale dei due if fa riferimento l'else.

```
Condition:
  'if' '(' condition=BooleanExpression ')'
    then=Expression
    (=>'else' else=Expression)?
;
```

Se la regola relativa è definita in questo modo (dove Condition è un sottotipo di Expression), il parser controlla se dopo l'espressione del then è presente la keyword "else", in tal caso l'espressione che segue è strettamente relativa alla Condition in esame e non a quella che la ha chiamata (nonostante anche la chiamante si aspettasse un else opzionale dopo l'espressione then).

Utilizzando "->" al posto di "=>" si indica al parser di controllare solo il token successivo per prendere la decisione, riducendo il lookahead e migliorando le prestazioni del parser e il comportamento fornito dall'IDE (es. del content assist).

Xtext usa un generator per produrre alcuni componenti come parser, serializer, classi per il content assist, modello Ecore, oltre che per realizzare files come plugin.xml e MANIFEST.MF, che contengono informazioni sul progetto. Il generatore si appoggia a MWE2, un DSL utilizzato per configurarlo. MWE2 instanzia classi Java, che permettono di eseguire una certa configurazione tramite metodi setter e adder, in base alla struttura ad albero che viene fornita.

Xtext permette di gestire diversi aspetti runtime del linguaggio programmandoli direttamente a livello di codice tramite apposite API:

- code generation/compilation: permette di scrivere un interprete che esamina l'AST producendo un determinato comportamento o di tradurre il linguaggio generato in un altro linguaggio di programmazione o in files di configurazione;

- validation e scoping: consente di gestire le informazioni da trasmettere all'utente mentre lavora con il linguaggio, ad esempio notificare errori nel caso in cui una variabile venga utilizzata al di fuori del suo scope;

- serialization: permette di trasformare il modello EMF in una rappresentazione testuale;

- formatting (o pretty printing): è usato per migliorare la leggibilità del documento indentando i blocchi di codice secondo una loro suddivisione logica;

- encoding: è usato per la scelta del character set descrivendo il modo in cui i caratteri sono codificati in bytes e vice versa, uno standard famoso è la codifica UTF-8;

- unit testing: consente la scrittura di tests automatizzati per il controllo del prodotto, aumentandone la qualità e la manutenibilità.

Xtext permette inoltre di configurare ed istruire i tools che gestiscono gli aspetti dell'IDE (un comportamento di default è già assegnato in automatico da Xtext), per facilitare il più possibile l'uso del linguaggio al programmatore:

- label provider: per presentare all'utente gli elementi del modello nelle diverse parti dell'interfaccia grafica (nella vista dell'outline, negli hyperlinks, nei suggerimenti, nelle finestre di ricerca..);

- content assist: per aiutare l'utente con suggerimenti riguardanti gli elementi accettabili in un determinato punto del codice;

- quick fixes: fanno riferimento alle validations e forniscono suggerimenti che permettono di risolvere l'errore o il warning in un possibile modo corretto;

- syntax coloring: oltre a strumenti come content assist e code formatter, il syntax coloring permette di migliorare la leggibilità del codice colorando differentemente keywords, commenti, stringhe, identificatori, etc.;

- rename refactoring: permette di fare un safe refactor (preceduto da validation e preview) degli elementi del linguaggio;

- template proposals: si possono definire alcune proposte di template di default o lasciare all'utente la possibilità di definire le proprie.

5.4 Esempio

In questo esempio si ipotizza di lavorare in ambiente Eclipse e si vuole realizzare un linguaggio di programmazione per la gestione dei dati anagrafici degli impiegati di un'azienda e del loro capo. Ovviamente l'esempio è facilmente rappresentabile in un'ottica object-oriented e direttamente in Java, senza scomodare Xtext. Nel caso in cui, però, si dovessero introdurre nuovi operatori e concetti, potrebbe essere sensato scrivere un linguaggio ad hoc (si veda ad es. SQL, DSL scritto con lo scopo di interrogare DataBases relazionali).

In particolare, si vuole che il linguaggio prodotto rispecchi queste specifiche:

```
typedef String
typedef Integer
typedef Date mapsto java.util.Date

entity Person {
  String name
  String surName
  Date birthDay
  Address home
  Address work
}

entity Boss extends Person {
  Person* employees
}
```

```
entity Address {
  String street
  String number
  String city
  String ZIP
}
```

Dove “Person* employees” indica che employees è un campo dell’entità Boss che contiene più Persons (in Java per rappresentare questo concetto si può fare uso di un array o di una lista, ad es.).

Si definisce per prima cosa la grammatica:

```
grammar org.xtext.example.Entity
    with org.eclipse.xtext.common.Terminals
generate entity
    "http://www.xtext.org/example/Entity"
```

se ne specifica il nome completo e il fatto che utilizza i concetti della super grammatica Terminals, e si definisce poi il nome e il namespace URI della grammatica.

```
Model:
  (types+=Type)*;

Type:
  TypeDef | Entity;

TypeDef:
  "typedef" name=ID ("mapsto"
                     mappedType=JAVAID)?;

JAVAID:
  name=ID("." ID)*;
```

```
Entity:
  "entity" name=ID ("extends"
                superEntity=[Entity])?
  "{"
    (attributes+=Attribute)*
  "}";

Attribute:
  type=[Type] (many?="*")? name=ID;
```

Model rappresenta l'entry point della grammatica perché è il primo elemento che compare dopo la dichiarazione della stessa.

Il modello è composto da più tipi ("*") che possono essere di tipo TypeDef o Entity. Il primo rappresenta la definizione di un nuovo tipo che può opzionalmente ("?") essere mappato in una sequenza di ID separati da punti (JAVAID), dove ID è definito nelle regole della grammatica Terminals. Una entità è identificata da un nome e può opzionalmente estenderne un'altra (si noti l'utilizzo del cross-ref "[Entity]"). All'interno di "{ }" possono essere specificate più ("*") liste ("+=") di attributi.

Un attributo ha un tipo (che può essere un TypeDef o una Entity, grazie all'utilizzo del cross-ref), può essere multiplo (simbolo "*") ed ha ovviamente un nome che lo identifica.

Lanciando il file "GenerateEntity.mwe" (automaticamente generato) come MWE Workflow, viene generata l'intera infrastruttura per il DSL descritto: parser, serializer, modello Ecore ed editor.

Lanciando poi il progetto come Eclipse Application, viene aperta una nuova istanza di Eclipse, in cui si può lavorare con il linguaggio appena definito come con un qualsiasi altro linguaggio.

Modificando il file in formato ".mwe2" si possono modellare certi comportamenti specifici. Ad esempio la riga

```
//composedCheck =
        "org.eclipse.xtext.validation.NamesAreUniqueValidator"
```

è commentata e ciò significa che, quando si dichiarano due elementi (che fanno riferimento alla stessa regola) con lo stesso nome, non si ha un errore finché uno dei due non viene riferito (cioè usato) nel codice. Se invece si decommenta la riga, si ha un errore non appena i due elementi vengono dichiarati con lo stesso nome: si realizza quindi l'unicità dell'identificativo.

Finora è stato realizzato solo un comportamento di controllo di correttezza del testo inserito, oltre al supporto automaticamente fornito dal framework, come il content assist già funzionante. Per definire il funzionamento della linguaggio si può creare un nuovo progetto che, caricando la grammatica, definisca in Java il funzionamento degli operatori e delle keywords introdotte.

Nello specifico di questo esempio, si potrebbe definire il comportamento di "typedef" e "mapsto" (magari associare il campo "name" ad una classe java il cui QualifiedName sia rappresentato da JAVAID, se presente, o cercare il corrispondente tipo di dato), così come quello di "*" (array o lista).

Si possono inoltre fornire meccanismi comodi al programmatore, che realizzano funzionalità semplici e generali. Ad esempio viene qui proposto un modo per la generazione automatica di getters e setters di ogni campo di una entità, che viene rappresentata come classe a sé stante.

Nel package "generator" si trova la classe "Generator.xtend", usata per generare codice sia in uno scenario stand alone che nell'ambiente di Eclipse. Dato un elemento (es. Entity), si trovano tutte le sue occorrenze nel file di linguaggio scritto dall'utente e, per ciascuna, si chiama il meccanismo di generazione di codice.

```
class EntityGenerator implements IGenerator {

  @Inject extension IQualifiedNameProvider

  override void doGenerate(Resource resource,
                    IFileSystemAccess fsa) {
    for(e: resource.allContents.toIterable.
                            filter(Entity)) {
       fsa.generateFile(e.fullyQualifiedName.
         toString("/") + ".java", e.compile)
    }
  }
```

Si itera per ogni risorsa che sia una entità ("filter(Entity)") e, per ogni occorrenza, si genera la classe Java con il nome corrispondente. Per conoscere il nome si usa il servizio di IQualifiedNameProvider, iniettando il suo codice all'interno del Generator.

A questo punto viene scritto il codice che dovrà far parte della classe Java, definito dal metodo compile chiamato nel generateFile.

```
  def compile(Entity e) '''
    «IF e.eContainer.fullyQualifiedName !=
                                        null»
      package
         «e.eContainer.fullyQualifiedName»;
    «ENDIF»

    public class «e.name»
    «IF e.superEntity != null»
         extends
         «e.superEntity.fullyQualifiedName»
    «ENDIF»
    {
      «FOR a : e.attributes»
        «a.compile»
      «ENDFOR»
    }
  '''
```

Il codice, scritto in linguaggio Xtend, definisce, se possibile, il nome del package in cui è contenuta la classe, dichiara la classe indicando eventualmente da quale questa estende e, per ogni attributo definito nel linguaggio, chiama nuovamente un metodo compile:

```
  def compile(Attribute a) '''
    private «a.type.fullyQualifiedName»
                                    «a.name»;

    public «a.type.fullyQualifiedName»

    get«a.name.toFirstUpper»() {
      return «a.name»;
    }
    public void set«a.name.toFirstUpper»(
                «a.type.fullyQualifiedName»
                «a.name») {
      this.«a.name» = «a.name»;
    }
  '''
}
```

Con l'ultima parentesi graffa si chiude la classe EntityGenerator. Il secondo metodo "compile" accetta un attributo "a" in ingresso e scrive nella classe Java la dichiarazione di un campo con il tipo e il nome specificati dall'utente, seguita dalla definizione dei metodi getter e setter per tale campo.

Infine, è possibile scrivere tests per monitorare il comportamento realizzato e, in fase di sviluppo, per controllare che successive modifiche continuino a soddisfare i tests già passati.

6. Conclusioni

Finora sono state presentate le caratteristiche fondanti di ogni singolo tool, supportate anche con esempi esplicativi. In questo ultimo capitolo si vogliono confrontare i tre strumenti, ponendo l'accento su punti forti e problemi noti di ciascuno.

Alcuni aspetti da tenere in considerazione sono: facilità di primo utilizzo (favorita anche dalla somiglianza ad altri linguaggi o standards già esistenti), qualità e livello di dettaglio della documentazione, facilità di reperibilità di materiale ed esempi online (documentazione inclusa), espressività del linguaggio, funzionalità messe a disposizione, integrazione con altri linguaggi ed ambienti e presenza di forums e comunità attivi.

Innanzitutto occorre notare che i tre sistemi presentati sono gratuitamente scaricabili dal web, al contrario di altri competitors forniti a pagamento. Nel presentare vantaggi e svantaggi che ogni tool può offrire nei confronti degli altri, è stato deciso di suddividere il capitolo in sottocapitoli dedicati.

6.1 JavaCC

È lo strumento più immediato da utilizzare grazie alla sintassi della grammatica simile a Java e ai concetti offerti. Purtroppo si integra solo con i linguaggi Java, C++ (dalla versione 6) e JavaScript (se associato all'utilizzo di GWT, Google Web Toolkit).

Fa uso di strumenti esterni (JJTree) per esaminare il parse tree, che forniscono comunque un valido supporto con il pattern visitor.

Una caratteristica decisamente positiva è rappresentata dal fatto che i prodotti di JavaCC non si appoggiano a dipendenze jar esterne, ma sono direttamente eseguibili.

Ciò permette di evitare problemi di versioning e di avere un codice più leggero e portabile. Uno svantaggio relativo si ha, però, quando si hanno più parsers per uno stesso progetto: onde evitare conflitti occorre dedicare un package per ogni parser.

Anche a causa di queste limitazioni, JavaCC è preferito per progetti relativamente semplici.

La documentazione fornita è ben dettagliata e copre moltissimi casi d'uso, ma occorre leggerla tutta per poter creare progetti di un certo spessore perché non è immediata e ben organizzata. Un valido supporto è dato anche dai forums messi a disposizione; è inoltre possibile acquistare un libro guida per la specifica della grammatica.

Un aspetto negativo che si incontra lavorando con JavaCC è l'utilizzo di codice inline, questo rende meno portabile tra i linguaggi target la grammatica realizzata (inizialmente l'unico linguaggio target era Java e quindi questo problema era meno evidente) e non permette di separare gli aspetti riguardanti la semantica da quelli riguardanti il comportamento del linguaggio. Inoltre non è permesso importare grammatiche esterne.

L'ultima release (JavaCC 6.0.1) risale all'8 Marzo 2014, quindi attualmente non c'è uno sviluppo attivo di questo tool.

6.2 ANTLR

È il tool che offre più portabilità grazie al codice generato in diversi linguaggi, ma non è il più semplice da imparare ed utilizzare. ANTLR offre più funzionalità di JavaCC, come ad es. la possibilità di importare grammatiche esterne, e la sua sintassi è molto simile alla notazione EBNF e perciò facilmente utilizzabile.

Occorre porre attenzione a commenti e descrizioni trovate in rete per quanto riguarda il supporto alla gestione di parse trees, infatti, dopo l'uscita della versione 4, questo è diventato molto più agevole grazie all'utilizzo dei patterns Observer e Visitor. Il tool fornisce inoltre una GUI per la visualizzazione grafica dell'albero di parsing, molto comoda in fase di produzione.

I progetti realizzati in ANTLR utilizzano diversi jars esterni (come antlr.jar acceduto a runtime) e sono quindi più pesanti e lenti da eseguire (durante il caricamento delle dipendenze). Ciò può creare conflitti tra jars importati dal proprio progetto e quelli automaticamente inclusi da ANTLR, oltre che a problemi di versioning.

La documentazione gratuita fornisce un buon "get started" per iniziare subito con piccoli progetti, però non è adeguatamente dettagliata e fa spesso riferimento al libro di supporto acquistabile separatamente.

Due punti a favore sono certamente rappresentati dalla possibilità di integrazione con l'ambiente Eclipse tramite l'apposito plugin e dal fatto che il codice scritto nel linguaggio target è situato in files diversi da quelli di specifica della grammatica.

Viene attivamente sviluppato e la versione più recente è ANTLR 4.5.1, rilasciata il 15 Luglio 2015.

6.3 Xtext

È certamente lo strumento più potente e completo tra i tre ed offre la possibilità di produrre un linguaggio in output con tutto il supporto che può fornire Eclipse.

Questo stretto contatto con Eclipse rappresenta però una lama a doppio taglio: da una parte permette di integrare perfettamente il proprio DSL in un ambiente molto noto, attivamente sviluppato e ricco di utilità per l'utente, d'altra parte, nonostante sia possibile, rende molto difficile lo sviluppo e l'utilizzo del proprio linguaggio al di fuori dell'ambiente Eclipse, anche a causa delle innumerevoli dipendenze utilizzate. Infatti dei tre è quello che fa utilizzo del maggior numero di jars e classi esterne e che quindi crea progetti più pesanti e lenti da caricare. L'overhead prodotto assume dimensioni notevoli se si considera l'uso di librerie, di MWE2 e la creazione di progetti addizionali associati a quello in costruzione.

Inoltre, tutte le funzionalità offerte rendono difficile il primo approccio con questo potente strumento, nonostante la grammatica sia semplice da definire, grazie anche all'introduzione di diversi concetti utili come, come gli unordered groups (difficilmente implementabili con gli altri due tools presentati).

Per quanto riguarda la documentazione, occorre notare che i forums sono particolarmente attivi e che questa è facilmente individuabile e ben organizzata. Purtroppo è però difficile realizzare comportamenti che non sono ben documentati con esempi, anche a causa delle strette integrazioni con altri linguaggi (come Xtend). Nonostante ciò, uno sviluppatore ferrato nel campo delle grammatiche e nella scrittura di estensioni per Eclipse può trovare veramente grandi vantaggi dall'utilizzo di Xtext, perché il prodotto risultante è decisamente più potente e funzionale di quello realizzato con gli altri due tools.

Una limitazione è rappresentata dal fatto che Java è l'unico linguaggio supportato, anche se grazie all'utilizzo di altri strumenti (come Xpand) è possibile scrivere templates per altri linguaggi.

Un notevole vantaggio offerto da Xtext è la separazione delle fasi di compilazione e validazione, oltre alla possibilità di personalizzare i messaggi di errore e warning.

Inoltre mette a disposizione alcuni meccanismi comodi al programmatore, che gli altri due tools non supportano: una volta definito il DSL, nello scrivere codice in tale linguaggio, hanno certamente un effetto facilitante e produttivo l'utilizzo di syntax coloring, code completion, code formatter e rename refactoring, per citarne alcuni.

Permette anche di importare grammatiche esterne con un meccanismo simile all'inheritance del modello object oriented.

Infine è certamente un fattore positivo il fatto che sia in costante aggiornamento: l'ultima release è del 30 Luglio 2015 (Xtext 2.8.4) e la prossima versione è prevista per il 28 Ottobre (Xtext 2.9.0).

6.4 Riepilogo

Quelli presentati sono solo tre dei possibili strumenti per lo sviluppo di linguaggi di programmazione e linguaggi domain specific. Infatti esistono diversi altri tools per la realizzazione di linguaggi e DSLs, tra cui:

- Spoofax: piattaforma per scrivere linguaggi domain specific integrabile con Eclipse;

- MPS di JetBrains: ambiente per la definizione di linguaggi, workbench e IDE;

- SableCC: generatore di compilers o interpreti in Java;

- Coco/R: altro generatore di compilers che produce scanner e parser per un linguaggio definito in una grammatica EBNF di tipo LL.

Un DSL fornisce una notazione su misura per il dominio applicativo di un certo problema, permette di produrre un linguaggio semplice da utilizzare anche per chi non ha grandi conoscenze di programmazione in generale e, sopratutto, offre vantaggi di produttività e prestazioni.

Per queste ragioni i DSLs sono sempre più largamente utilizzati e il trend degli ultimi anni sembra confermare questa ipotesi. In particolare, JavaCC è stato largamente utilizzato soprattutto negli anni passati, mentre le tendenze attuali propendono più per tools più moderni come ANTLR e Xtext.