UNIVERSITÀ DEGLI STUDI DI PARMA
Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica
Tesi di Laurea Triennale

Definizione di un Linguaggio di Programmazione Imperativo e Realizzazione di un Prototipo mediante Xtext

Definition of an Imperative Programming Language and Implementation of an Xtext Prototype

Relatore
Chiar.mo Prof. Federico Bergenti

Candidato
Manuel Rinaldi

Anno Accademico 2012/2013

Ai miei genitori,
a Cecilia.

Ringraziamenti

Il mio più grande ringraziamento va ai miei genitori e alla mia ragazza, per avermi supportato in ogni modo in questi anni di studio. Un sentito ringraziamento anche ai miei compagni di corso che hanno studiato e scherzato con me condividendo la stessa passione e facendo il tifo per me. Un grazie particolare al professor Federico Bergenti per la sua grande disponibilità e professionalità, nonché per l'aiuto offertomi e per i preziosi suggerimenti sulla mia carriera universitaria e su questo lavoro di tesi. Infine, un pensiero ai miei nonni che fanno ancora fatica a capire "che cosa divento adesso", a tutti i miei amici e a Denise.

Indice

Prefazione

La scelta di un linguaggio di programmazione da utilizzare per sviluppare le proprie applicazioni è di fondamentale importanza, poiché ogni linguaggio offre caratteristiche che differiscono tra loro a volte anche in modo sostanziale. Oltre a caratteristiche cosiddette *esterne*, come ad esempio portabilità, diffusione ed integrabilità, si tiene conto anche di aspetti intrinseci. In generale i linguaggi moderni offrono una gamma piuttosto ampia di funzionalità, ma l'eterogeneità dei campi d'applicazione suggerisce il bisogno di poter sviluppare un linguaggio creandolo appositamente per quell'ambito specifico. In effetti, un programmatore potrebbe voler determinare a priori caratteristiche del linguaggio che andrà ad utilizzare, con particolare enfasi sulla sintassi.

Il principale obiettivo di questo lavoro di tesi è quello di sperimentare il tool *Xtext*, un framework per lo sviluppo di linguaggi di programmazione o di *DSLs* ossia *Domain Specific Languages*. Partendo da una grammatica in una forma simile a quella di Backus-Naur estesa (*EBNF*), *Xtext* mette a disposizione principalmente un parser, un modello a classi basato su *EMF* per l'albero di sintassi astratta e un editor basato su Eclipse ed altamente integrato con il linguaggio stesso. A questi si aggiungono ulteriori strumenti che permettono di personalizzare come meglio si crede il proprio linguaggio.

Se la sintassi viene specificata tramite una grammatica, la semantica è definita invece attraverso la generazione di codice in un linguaggio già noto, come ad esempio Java, C, C++, Python ed altri. La scelta più comune è quella di generare codice Java, scelta non affatto casuale poiché Java stesso è stato progettato con lo scopo di essere indipendente dalla piattaforma sottostante grazie all'implementazione della *Java Virtual Machine*. La sua grande portabilità, in congiunzione con la facilità di riuso del codice grazie all'approccio *object-oriented*, ha infatti contribuito alla diffusione su larga scala.

Il linguaggio che è stato implementato prende il nome di FALL1, acronimo di *Flexible Agent Level Language versione 1*, ed è un linguaggio di programmazione concepito inizialmente per lo scambio di informazioni tra

agenti e per il calcolo distribuito. Ad oggi il linguaggio, che si trova in uno stato embrionale, presenta alcune caratteristiche base per la scrittura di semplici programmi ed è strutturato in modo da facilitarne la comprensione da parte di chi si approccia per la prima volta alla scrittura di codice. In particolare FALL1 adotta una struttura a blocchi basata sull'indentazione del codice, simile a quella offerta da Python. Inoltre è presente la manipolazione di alcuni dati primitivi senza limiti di rappresentazione, sfruttando le classi Java `BigInteger` e `BigDecimal`, oltre alla possibilità di avere a disposizione liste ed insiemi *built-in*. La versione di *Xtext* utilizzata è la 2.3.1.

Il lavoro di tesi è strutturato come segue.

Il capitolo 1 è dedicato ad una introduzione generale sui linguaggi formali, le grammatiche che generano tali linguaggi ed in particolare i linguaggi LL(*) con qualche informazione su come si comporta un parser LL(*).

Il capitolo 2 descrive le caratteristiche principali dei tool di sviluppo utilizzati e permetterà di capire come si può definire un linguaggio e cosa si andrà ad ottenere.

Il capitolo 3 riguarda il linguaggio sviluppato in cui verrà descritta la sintassi e la semantica così come altri aspetti tenuti in considerazione. Inoltre ci si soffermerà su alcune difficoltà incontrate durante lo sviluppo in merito alla realizzazione di determinate caratteristiche.

Il capitolo 4 è dedicato alle considerazioni personali riguardanti il lavoro svolto e alle caratteristiche che potranno essere sviluppate nelle successive versioni del linguaggio.

Infine, l'appendice A contiene qualche esempio di programma scritto nel linguaggio sviluppato.

Capitolo 1

Linguaggi LL(*)

La sintassi di un linguaggio, ossia i costrutti che effettivamente si possono scrivere, può essere espressa attraverso una grammatica. In questo capitolo si vuole dare una visione generale del concetto di grammatica e del linguaggio da essa generato, per poi arrivare a parlare della soluzione adottata dal parser utilizzato da *Xtext*, cioè ANTLR.

1.1 Grammatiche CF, concetti generali

Un parser è sostanzialmente un programma che ha il compito di analizzare un flusso di simboli in input precedentemente suddiviso in unità atomiche chiamate *token*, con lo scopo di verificare se tale sequenza rispetta le regole fornite da una grammatica specificata. Il risultato finale quindi della fase di parsing altro non è che la generazione di un *parse tree*. L'analisi sintattica può essere affrontata con approccio *top-down*, come fanno ad esempio i parser LL (leggono l'input da sinistra a destra cioè ***L**eft-to-right*, e ad ogni passo si ha una ***L**eftmost derivation*), oppure *bottom-up* tramite l'uso di parser LR (lettura sempre ***L**eft-to-right*, ma con una ***R**ightmost derivation*). Da qui in poi faremo riferimento solamente all'approccio LL e descriveremo la strategia di parsing LL(*) su cui si basa il generatore di parser ANTLR trattato nel capitolo 2.1.

Una *grammatica libera dal contesto* (*CF*) è una quadrupla $G = \langle V, T, P, S \rangle$ dove:

- V è un insieme finito di simboli non terminali
- T è un insieme finito di simboli terminali tale che $V \cap T = \emptyset$
- P è un insieme finito di produzioni nella forma $A \to \alpha$ con:

  - $A \in V$
  - $\alpha \in (V \cup T)^*$

- $S \in V$ è il simbolo iniziale

Data una grammatica libera dal contesto $G = \langle V, T, P, S \rangle$, si può dare una definizione di linguaggio generato da tale grammatica in termine di alberi di derivazione, o parse tree. Diremo che un albero è un parse tree per G se:

1. ogni nodo ha un'etichetta presa in $V \cup T \cup \{\varepsilon\}$
2. l'etichetta della radice dell'albero appartiene a V
3. ogni nodo interno ha etichetta appartenente a V
4. se un nodo n ha etichetta A ed $n_1, \ldots, n_k$ sono, ordinatamente da destra a sinistra, i nodi figli di A ed hanno rispettivamente etichette $X_1, \ldots, X_k$, allora $A \to X_1 \ldots X_k \in P$
5. se un nodo n ha etichetta ε, allora n è una foglia ed è l'unico figlio di suo padre

La stringa $\alpha \in (V \cup T)^*$ descritta dal parse tree costruito come sopra è la stringa che si legge da sinistra a destra con le etichette dei nodi foglia.

Se ad ogni passo viene espanso il simbolo non terminale che si trova più a sinistra, si parlerà di *leftmost derivation*, mentre al contrario se ad ogni passo si espande il non terminale più a destra avremo una *rightmost derivation*. Se per una grammatica CF esiste una stringa che ha più di un albero di derivazione con radice S, tale grammatica si dice ambigua [1]. La grammatica $G = \langle \{A\}, \{1\}, P, A \rangle$ con P definito come segue:

$$A \to A + A \mid 1$$

è ambigua, come dimostrato dalla figura 1.1

Sia ora una grammatica $G = \langle V, T, P, S \rangle$. Diremo che G è una grammatica ricorsiva sinistra, o *left-recursive* se esiste un simbolo non terminale A che deriva ad un certo punto una stringa in cui A stesso compare come simbolo più a sinistra. Questo si può verificare con una derivazione immediata del tipo:

$$A \to A\alpha \mid \beta$$

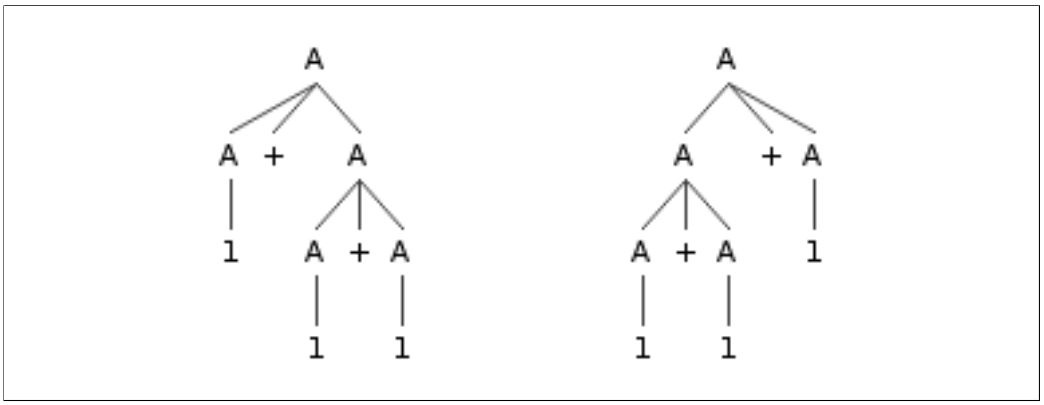

Figura 1.1: Due parse tree per la stringa 1+1+1.

con $\alpha, \beta \in (V \cup T)^*$ e β che non cominci con il simbolo A, oppure con una derivazione indiretta. Prendiamo come esempio di quest'ultima le seguenti produzioni:

$$A \to B\alpha \mid C$$
$$B \to A\beta \mid D$$

A partire dal simbolo non terminale A si può infatti avere la derivazione ricorsiva $A \overset{G}{\Rightarrow} B\alpha \overset{G}{\Rightarrow} A\beta\alpha \overset{G}{\Rightarrow} \ldots$

Le grammatiche left-recursive, prima di essere sottoposte al parsing tramite un parser LL, devono essere trasformate per rimuovere la ricorsione sinistra altrimenti si potrebbe avere un ciclo infinito durante una derivazione. Qualche esempio di eliminazione della ricorsione sinistra può essere trovato in [1, Lemma 6.19, p. 70] per un caso generale oppure in [2] per il caso particolare di *Xtext*. L'operazione di eliminazione della ricorsione sinistra può però contorcere la grammatica in modo significativo. Come vedremo nel capitolo 2.1 a proposito della specifica della sintassi, avremo la possibilità di utilizzare il tool ANTLR ed alcuni specifici costrutti che ci potranno aiutare a mantenere una buona leggibilità e generale controllo sulla grammatica.

1.2 Lookahead e backtracking

Data una sequenza di token in input, un parser si occupa di analizzare tale stringa verificando se essa segue o meno le regole di una grammatica precedentemente specificata. In questa fase viene generato normalmente un parse

tree che descrive la stringa. Nel caso in cui una regola abbia più alternative, il parser deve essere in grado in qualche modo di decidere quale sarà quella corretta da espandere. In alcuni casi tale decisione è semplice, mentre in altri sono necessarie tecniche appropriate come ad esempio l'utilizzo del lookahead che è la massima quantità di token ulteriori (ossia token che seguono quello correntemente letto) che un parser può utilizzare nella decisione.

L'uso del lookahead, oltre a facilitare la scelta per l'alternativa corretta, abbassa significativamente la complessità della scrittura del parser stesso. Un parser LL si dice LL(k) se utilizza k token di lookahead. Questo significa che nel prendere una decisione per una determinata stringa, il parser può avanzare fino a k token nell'input. Attraverso l'uso del lookahead e considerando grammatiche LL(k), è possibile realizzare parser predittivi per tali grammatiche, ossia in grado di decidere l'alternativa corretta solamente utilizzando i k token successivi. Ciò potrebbe però non essere sufficiente in caso di grammatiche di altro tipo per le quali, in combinazione alla tecnica del lookahead, può essere necessario utilizzare il backtracking. Questa tecnica consiste nel tornare ad un punto precedente all'espansione di un simbolo non terminale nel momento in cui ci si accorge di aver scelto un'alternativa non corretta. Tali parser vengono chiamati ricorsivi discendenti con backtracking. Ad esempio supponiamo di voler effettuare il parsing della stringa in input *abc* considerando la seguente grammatica:

$$S \rightarrow aBc$$
$$B \rightarrow bc \mid b$$

In un primo momento il simbolo S viene espanso con i simboli aBc. A questo punto due alternative sono possibili per l'espansione del simbolo B. Supponendo che il parser scelga la prima alternativa, si arriva ad una stringa di non terminali *abcc* che non corrisponde a quella di input. É stato quindi generato un parse tree non corretto, ma riusciamo comunque ad intuire che se il parser avesse deciso di scegliere la seconda alternativa avremmo avuto un parse tree corrispondente alla stringa *abc*. A questo punto subentra il backtracking, il parser torna alla decisione precedente di quale alternativa scegliere e, avendo escluso la prima, procede con la seconda che si rivela poi quella corretta. In questo esempio non è stato detto nulla sulla presenza di lookahead, che avrebbe comunque facilitato le cose. In figura 1.2 sono rappresentati i due parse tree generati prima e dopo il backtracking.

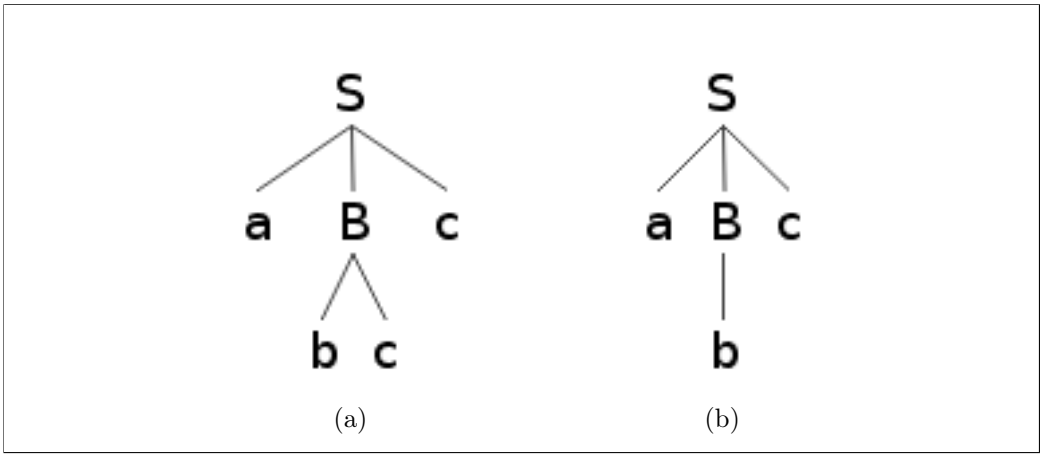

Figura 1.2: Due parse tree generati durante il parsing della stringa *abc*.

1.3 Parser LL(*)

Un parser LL(*) è sostanzialmente un parser LL con backtracking che può utilizzare un numero arbitrario di token di lookahead al posto di un numero fissato k. La decisione di quale k utilizzare avviene dinamicamente durante la fase di parsing fino ad arrivare, se necessario, all'uso del backtracking in base alla complessità delle decisioni e ai simboli in input.

Nella pratica però i parser LL(*) utilizzano in media uno o due token di lookahead utilizzando il backtracking solo occasionalmente. In figura 1.3a vediamo chiaramente come la media di token di lookahead sia minore di due anche per input di decine di migliaia di linee, mentre in figura 1.3b viene visualizzata nella colonna "Backtrack" la percentuale di backtracking effettuato su un totale di decisioni indicati nella colonna "decision events". Nonostante la potenza dei parser LL(*) copra anche l'ambito dei linguaggi dipendenti dal contesto (un sovrainsieme dei linguaggi CF), viene comunque mantenuta una grande espressività così come un buon livello di gestione degli errori cercando di minimizzare il più possibile il non determinismo (ossia le decisioni di effettuare backtracking). Il problema principale rimane quello dell'efficienza essendo i parser LL(*) di tipo ricorsivo-discendente, problema che grazie alla potenza dei calcolatori attuali può essere nella maggior parte dei casi tralasciato.

La strategia di fondo per i parser LL(*) è quella di costruire un'espressione regolare e quindi alla fine un *automa a stati finiti* (*DFA*) per ogni simbolo non terminale della grammatica, in modo da poter identificare quale sarà

Grammar	Input lines	parse-time	n	avg k	back. k	max k
Java1.5	12,394	471ms	111	1.09	3.95	114
RatsC	37,019	1,375ms	131	1.88	5.87	7,968
RatsJava	12,394	527ms	78	1.85	5.95	1,313
VB.NET	4,649	339ms	166	1.07	3.25	12
TSQL	794	164ms	309	1.08	2.63	20
C#	3,807	524ms	146	1.04	1.60	9

(a)

Grammar	Can back.	Did back.	decision events	Back-track	Back. rate
Java1.5	19	16	462,975	2.36%	45.22%
RatsC	30	24	1,343,176	16.85%	65.27%
RatsJava	8	7	628,340	14.07%	74.68%
VB.NET	6	3	109,257	0.46%	20.84%
TSQL	29	19	17,394	3.38%	27.01%
C#	24	19	141,055	3.68%	40.22%

(b)

Figura 1.3: Test di lookahead medio e percentuali di backtracking.

l'alternativa corretta da espandere. Se l'analisi non riesce a trovare un DFA adatto per un certo non terminale, si passa all'esecuzione del backtracking. Prendiamo come esempio una grammatica come la seguente, con omessa la regola `Expr`:

```
1 S:   ID
2    | ID '=' Expr
3    | 'unsigned'* 'int' ID
4    | 'unsigned'* ID ID
5 ;
```

Analizziamo ora la situazione. Al momento della decisione di quale alterna-

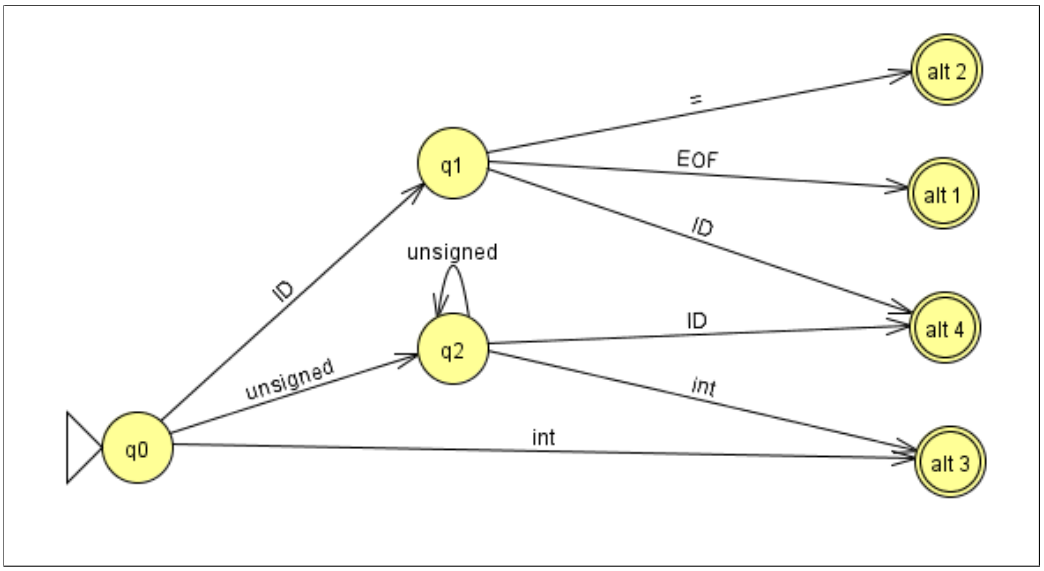

Figura 1.4: DFA per S. L'etichetta `alt n` significa n-esima alternativa.

tiva scegliere per S, la stringa di input viene elaborata dal DFA in questione fino al raggiungimento di uno stato accettante che indica quale strada percorrere. Supponendo di avere come input la stringa `int x`, al momento della lettura del token `int` l'alternativa scelta è l'unica possibile ossia la numero 3. In questo caso il valore fissato di lookahead sarà $k = 1$. Prendendo invece come input la stringa `T x` si ha che al momento della lettura del token `T`, ossia un ID, le alternative possibili possono essere la 1, la 2 o la 4. Per scegliere quella corretta, cioè la 4, si ha quindi bisogno di un lookahead fissato pari a $k = 2$. Il caso di lookahead arbitrario si verifica invece se in input si ha il token `unsigned`, poiché bisognerà scandirne un numero non precisato prima di arrivare al token che deciderà l'alternativa 3 piuttosto che la 4.

Il linguaggio generato dal simbolo non terminale `S` è regolare, perciò in questo caso si può costruire il relativo automa a stati finiti che sarà quello di figura 1.4.

In alcuni casi però, soprattutto in caso di regole ricorsive (ma non ricorsive sinistre), non si potrà costruire un DFA poiché il linguaggio generato da un certo simbolo non terminale sarà context free ma non regolare. In questo caso la soluzione proposta dal parser di ANTLR è quella di costruire comunque un DFA con un certo numero aggiuntivo di stati che permette in molti casi di evitare il backtracking, il quale si verificherà solamente come ultima opzione [3].

Capitolo 2

ANTLR e Xtext

ANTLR è l'acronimo per *ANother Tool for Language Recognition* ed è sostanzialmente un generatore di parser che fa uso del parsing LL(*). A partire da una grammatica context-free ma non ricorsiva sinistra, ANTLR produce un parser a discesa ricorsiva (meno efficiente ma di più facile comprensione rispetto ad un parser *table driven*) per il relativo linguaggio generato oltre a meccanismi per la visita del parse tree. In questo capitolo si vogliono illustrare gli strumenti per lo sviluppo del linguaggio descrivendone in breve le principali caratteristiche ed il funzionamento.

2.1 ANTLR

ANTLR prende in input qualsiasi grammatica context-free ad eccezione di quelle ricorsive sinistre, ed è in grado di identificare staticamente alcune ambiguità e produzioni irraggiungibili. La grammatica viene specificata in una forma simile a quella EBNF, arricchita con costrutti ausiliari che permettono di aumentare il livello di dettaglio. Essa si compone di una serie di regole nella forma generale e semplificata:

$$RuleName : rightSideRule;$$

La prima regola specificata viene chiamata *entry-rule* ed ha una funzione simile al simbolo iniziale di una grammatica.

2.1.1 Attributi e operatori

La parte destra di una regola può avere degli attributi che fungono da variabili per assegnamento. In effetti una regola può essere vista ad un livello astratto come una classe Java (ad essere rigorosi sarà un oggetto di tipo `EClass`), ed è

verosimilmente quello che *Xtext* genera e ciò su cui si lavorerà nelle successive fasi. Si possono distinguere tre operatori di assegnamento:

1. L'operatore =, cioè di assegnamento diretto utilizzato per variabili che devono contenere un solo elemento.
2. L'operatore +=, utilizzato per l'assegnamento multiplo, ad esempio quando si ha a che fare con delle liste.
3. L'operatore ?=, per l'assegnamento di valori booleani *true* o *false* rispettivamente se la parte destra viene consumata oppure no.

Inoltre sono presenti operatori per specificare le cardinalità, nel dettaglio si avrà:

1. L'operatore ? che indica nessuna o una ripetizione.
2. L'operatore + che indica una o più ripetizioni.
3. L'operatore * che indica zero o più ripetizioni.
4. Nessun operatore per indicare esattamente una ripetizione.

Nella fase di analisi lessicale, il flusso di caratteri deve essere trasformato in una sequenza di unità atomiche. Ogni unità viene chiamata token e forma un simbolo composto ben preciso e fondamentale per la successiva fase di analisi sintattica. ANTLR offre la possibilità di definire i token attraverso regole terminali, appositamente precedute dalla parola chiave `terminal`.

Le alternative di una regola vengono specificate con il simbolo '|' che permette di aumentare la leggibilità. Si prenda ad esempio una regola che specifichi l'insieme dei numeri naturali. Questa avrà una forma del tipo:

$$terminal\ NUMBER\ :\ '0' \mid '1' \mid '2' \mid \ldots$$

Per ovvie ragioni è impossibile specificare ogni numero, per cui possiamo sfruttare appositi costrutti di ANTLR come l'operatore di range '..' e le parentesi per il raggruppamento.

La regola precedente può quindi essere riscritta in questo modo:

$$terminal\ NUMBER\ :\ ('0' \mathinner{..} '9')('0' \mathinner{..} '9')*;$$

oppure nella forma più compatta:

$$terminal\ NUMBER\ :\ ('0' \mathinner{..} '9')+;$$

Altri simboli speciali di ANTLR sono il simbolo di *until* '`->`', il simbolo *wildcard* '.' e il simbolo di negazione '!'.

Il primo serve ad identificare una sequenza di caratteri delimitata da un simbolo di inizio e uno di fine come ad esempio i commenti di un codice sorgente contenuti fra i simboli '/*' e '*/': tutto quello che sta in mezzo farà parte del commento. Il secondo serve ad indicare un qualsiasi simbolo. La regola 'f' . 'o' identificherà quindi token come `foo`, `fio` oppure `f_o`. Allo stesso modo la regola 'f' .* 'o' si riferirà a tutto quello che inizia per 'f' e finisce per 'o' come ad esempio `foo`, `f123o`, `fo` eccetera.

Per disambiguare l'operatore wildcard dal simbolo 'punto', quest'ultimo andrà scritto fra apici o virgolette. Il simbolo di negazione viene invece utilizzato ovviamente per considerare input differenti da quello che si specifica dopo il simbolo stesso.

Alcune regole terminali di uso comune come ad esempio INT, ID, STRING, ML_COMMENT sono messe a disposizione nell'apposita grammatica di *Xtext* denominata `org.eclipse.xtext.common.Terminals`.

2.1.2 Riferimenti

Nella fase di specifica della grammatica è anche possibile creare dei riferimenti tra regole, funzionalità molto utile al programmatore. In questo modo si sfrutterà l'integrazione con l'editor di Eclipse per ottenere la navigabilità nel codice.

Si pensi ad esempio alla dichiarazione di una variabile e all'assegnamento di un valore ad essa in un punto del codice molto lontano dalla dichiarazione. Il programmatore potrebbe voler risalire il codice rapidamente fino ad arrivare alla dichiarazione della variabile. *Xtext* ha un comportamento di default per risolvere i riferimenti molto semplice ossia un riferimento è risolto visitando l'albero del modello fino ad arrivare alla prima istanza dell'oggetto referenziato. Talvolta questo comportamento può avere bisogno di essere modificato. Nel caso del linguaggio FALL1, ad esempio, si dovrà tenere presente della struttura a blocchi e per questo se ne implementerà una versione alternativa nella fase di *scoping*.

L'operazione di *cross-referencing* viene specificata con l'uso di parentesi quadre che contengono non il nome di una regola ma il nome della la sua `EClass`. Tale nome corrisponde di default effettivamente al nome della regola, ma bisogna tenere presente che *Xtext* ci permette di specificare esplicitamente l'`EClass` tramite la parola chiave `returns`.

Prendiamo come esempi le regole presenti nel listato 2.1. Come si può notare nella regola `Assignment` è presente un riferimento a `Variable`. Questo sta a significare che, al momento della creazione di un oggetto di tipo

```
1  Variable:
2     type=Type name=ID';'
3  ;
4
5  Assignment:
6     name=[Variable] '=' value=Value';'
7  ;
8
9  Value:
10    INT | STRING
11 ;
12
13 Type:
14    'integer' | 'string'
15 ;
16
17 terminal INT returns ecore::EInt: ('0'..'9')+;
18
19 terminal STRING :
20         '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') |
               !('\\'|'"') )* '"' |
21         "'" ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') |
               !('\\'|"'") )* "'"
22 ;
```

Listato 2.1: Riferimenti.

`Assignment` che ha nell'attributo `name` il valore `x`, un oggetto di `EClass Variable` che ha come valore dell'attributo `name` lo stesso `x` deve già essere stato istanziato, altrimenti si avrà a che fare con la segnalazione di un errore dovuto ad una *broken reference*.

La scelta di `name` come nome attributo non è casuale poiché si tratta di una parola chiave che in questo contesto permette di ottenere una certa interattività con il codice grazie ad un semplice *Ctrl+click* sul riferimento.

Un'altra cosa da notare è la regola terminale `INT`, che specifica tramite la parola chiave `returns` la sua `EClass`. Essa corrisponde in questo caso al tipo `ecore::EInt`, un tipo di dato messo a disposizione da *EMF* (paragrafo 2.2.1) per inizializzare i modelli relativi ai tipi di dati primitivi.

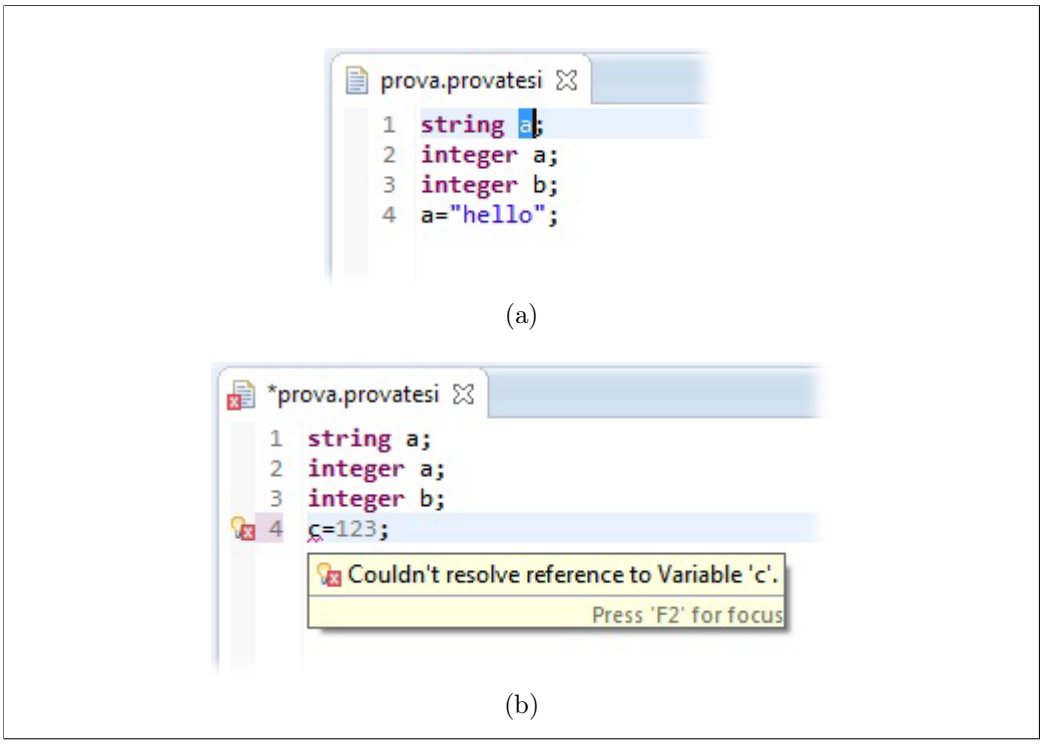

Figura 2.1: Comportamento dei riferimenti.

Un esempio di codice corretto, secondo la grammatica del listato 2.1, può essere il seguente:

```
1 string a;
2 integer a;
3 integer b;
4 a="hello";
```

In figura 2.1a viene rappresentato ciò che succede utilizzando il comando Ctrl+click sulla parte sinistra dell'assegnamento, mentre in 2.1b si ha un esempio di broken reference.

```
Variable:
   {TypedVariable} type=Type name=ID ';' |
   {UntypedVariable} name=ID ';'
;

Type:
   'integer' | 'string' | 'boolean'
;
```

Listato 2.2: Azioni semplici.

2.1.3 Azioni

La creazione di un oggetto a partire da una regola avviene normalmente con una politica *lazy* al momento del primo assegnamento. Il suo tipo viene generalmente dedotto in base al nome della regola ma può anche essere reso esplicito mediante l'uso delle cosiddette azioni. Le azioni possono essere di due tipi, azioni semplici e azioni di assegnamento e si specificano mediante l'uso di parentesi graffe.

Il primo tipo di azione viene utilizzata soprattutto per migliorare la leggibilità di una grammatica e ci permette ad esempio di specificare che un certo tipo `A` è sottotipo di un altro tipo `B`.

Potremo quindi scrivere qualcosa di simile al codice riportato nel listato 2.2. Al momento del parsing della regola `Variable`, viene istanziato un oggetto di tipo `TypedVariable` o `UntypedVariable` a seconda della scelta del parser. Tale tipo estenderà `Variable`. Senza l'uso delle azioni avremmo invece avuto un unico caso con l'istanziazione di un oggetto di tipo `Variable`.

L'azione di assegnamento è invece più interessante e ci viene in aiuto soprattutto nella scrittura delle espressioni. Quello che ci permette di fare è ad esempio definire la sintassi delle operazioni tra espressioni, trasformando una grammatica precedentemente trattata con *fattorizzazione sinistra* e riscrivendo l'albero di sintassi astratta tramite la parola chiave `current`.

Prendiamo come esempio la grammatica fattorizzata a sinistra del listato 2.3. Quello che succede è che al momento del parsing ad esempio della regola `MultiplicativeExpression`, se è presente l'operatore '*' viene creato un oggetto `Operation` a cui viene assegnato come attributo `leftOperand` la `Expression` creata durante il parsing della regola `PrimaryExpression`.

```
1  Expression returns Expression:
2     {Operation} leftOperand=MultiplicativeExpression (op='+'
          rightOperand+=MultiplicativeExpression)*
3  ;
4
5  MultiplicativeExpression returns Expression:
6     {Operation} leftOperand=PrimaryExpression (op='*'
          rightOperand+=PrimaryExpression)*
7  ;
8
9  PrimaryExpression returns Expression:
10    '(' Expression ')' |
11    {IntLiteral} value=INT
12 ;
```

Listato 2.3: Riscrittura dell'AST.

Si consideri però l'espressione '100'. Con la prima forma della nostra grammatica l'`Expression` iniziale sarebbe un oggetto di tipo `Operation` che ha come `leftOperand` un altro oggetto di tipo `Operation` che a sua volta ha come `leftOperand` un oggetto di tipo `IntLiteral` con il valore 100. L'albero di sintassi astratta per questa espressione ha un overhead di informazioni che non sono di aiuto e risultano inutili.

Utilizzando le azioni di assegnamento possiamo riscrivere il tutto come indicato nel listato 2.4. Nella seconda forma, la nostra `Expression` sarà semplicemente un oggetto di tipo `IntLiteral` con il valore 100.

In figura 2.2a si può notare l'AST relativo alla prima forma della grammatica mentre in figura 2.2b quello relativo alla seconda forma.

2.1.4 Predicati sintattici

Un predicato sintattico viene espresso tramite l'operatore freccia '`=>`' e serve ad "indirizzare" il parser verso una scelta piuttosto che un'altra. Formalmente un predicato sintattico è definito come:

$$A \rightarrow (A_i') \texttt{ => } \alpha_i$$

In pratica il simbolo A espande in α_i solo se l'input corrente rispecchia la sintassi descritta da A_i'.

```
1  Expression returns Expression:
2     MultiplicativeExpression ({BinaryOperation.leftOperand=current}
          op='+' rightOperand+=MultiplicativeExpression)*
3  ;
4
5  MultiplicativeExpression returns Expression:
6     PrimaryExpression ({BinaryOperation.leftOperand=current} op='*'
          rightOperand+=PrimaryExpression)*
7  ;
8
9  PrimaryExpression returns Expression:
10    '(' Expression ')' |
11    {IntLiteral} value=INT
12 ;
```

Listato 2.4: Riscrittura dell'AST.

Analizziamo ad esempio il noto problema del *dangling else*, ovvero di come gestire la seguente situazione:

```
if (condizione1)
   if (condizione2)
      istruzione1
   else
      istruzione2
```

Il comportamento che ci si aspetta è che il ramo `else` appartenga al costrutto `if` più vicino, come suggerito dall'indentazione del codice, ma potrebbe non essere così per la decisione del parser.

Per ovviare a questo inconveniente si può aggiungere un predicato sintattico che suggerisce al parser di controllare dopo il ramo `if` se la parola `else` è presente e, in caso affermativo, consumare immediatamente il token ed espandere quel ramo. In questo modo si ottiene il comportamento che ci si aspetta.

La regola riportata nel listato 2.5 risolve attraverso un predicato sintattico il problema del dangling else.

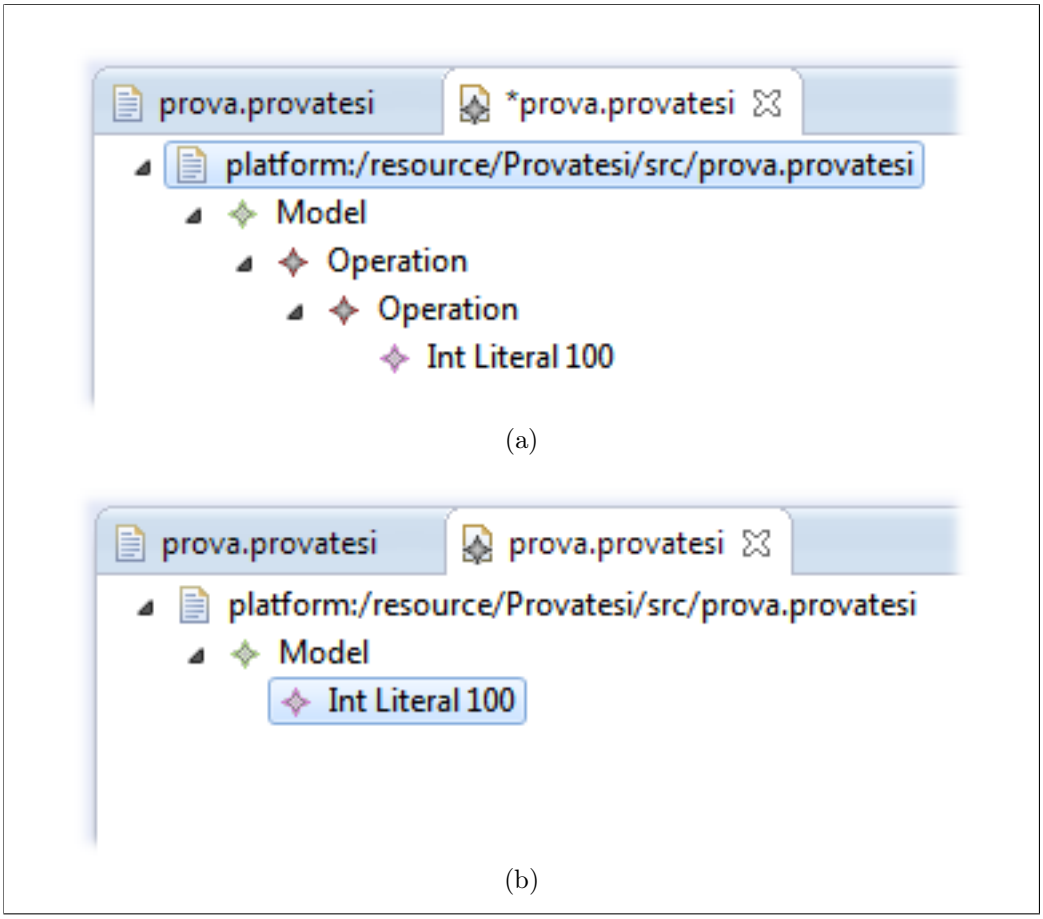

Figura 2.2: Due alberi di sintassi astratta per l'espressione '100'.

2.1.5 Predicati semantici

I predicati semantici sono espressioni booleane scritte nel linguaggio *host* specificate all'interno di parentesi graffe seguite dal simbolo '?' ed hanno lo scopo di introdurre regole semantiche anche all'interno della grammatica. Un predicato semantico è formalmente definito come:

$$A \rightarrow \{\pi_i\}?\ \alpha_i$$

Il suo significato è che il simbolo A espande ad α_i solo se il predicato π_i vale nell'istante in cui ci si trova.

Facciamo un esempio, supponendo di voler descrivere con una regola i numeri che hanno al più cinque cifre.

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

Listato 2.5: Regola per il dangling else.

Una possibilità potrebbe essere quella di definire tutte le alternative come ad esempio:

```
FiveDigitNumber:
   Digit |
   Digit Digit |
   Digit Digit Digit |
   Digit Digit Digit Digit |
   Digit Digit Digit Digit Digit
;
```

L'alternativa è utilizzare un predicato semantico come nell'esempio seguente:

```
FiveDigitNumber
@init { int N = 0; }
  :  (Digit { N++; } )+ { N <= 5 }?
;
```

Quello che succede è che, nel momento in cui il parser "entra" nella regola `FiveDigitNumber`, viene inizializzata una variabile contatore di nome `N` con il valore zero. Ogni volta che si raggiunge un `Digit`, il contatore viene aumentato di uno e, se il conteggio è maggiore di 5, viene prodotta dal parser una eccezione di tipo `FailedPredicateException`.

2.2 Xtext

Come già accennato in precedenza, *Xtext* è un framework open-source altamente configurabile utilizzato per sviluppare linguaggi di programmazione oppure *DSLs*. Tra le varie funzionalità offerte le principali, ossia quelle che ci permetteranno di ottenere effettivamente il nostro linguaggio, sono la scrittura della grammatica, lo *scoping*, il *validator* e il *generator*.

Per quanto riguarda la grammatica, *Xtext* sfrutta il generatore di parser ANTLR di cui abbiamo parlato nel capitolo 2.1, anche se con qualche modifica per meglio adattarsi al modello a classi EMF. Tuttavia, alcuni costrutti presenti in ANTLR come ad esempio i predicati semantici non saranno resi disponibili per la stesura della grammatica con *Xtext*.

In questo paragrafo daremo qualche informazioni circa le fasi successive alla scrittura della grammatica e ci occuperemo anche di aspetti meno importanti ma comunque utili che ci permetteranno di personalizzare il linguaggio in modo dettagliato. Per approfondimenti sulla documentazione di *Xtext* si veda [4].

2.2.1 EMF - Eclipse Modeling Framework

Il principale strumento su cui si basa *Xtext* prende il nome di *Eclipse Modeling Framework* ed è un framework di modellazione che ha come scopo principale l'unificazione di tre differenti strumenti di rappresentazione di un sistema:

- il codice Java
- il diagramma a classi UML
- il modello XML

L'insieme di informazioni raccolte tramite questi tre strumenti viene racchiuso in un unico costrutto che chiameremo *model* il quale altro non è che la rappresentazione mediante un *class diagram* UML del file di testo di cui viene fatto il parsing. Il model è in effetti la rappresentazione dell'albero di sintassi astratta (AST) ed è costituito da più `EObject` connessi tra loro. Ogni `EObject` è istanza di una `EClass` mentre un insieme di `EClass` verrà chiamato `EPackage`.

Siccome l'albero di sintassi astratta rappresenta effettivamente il model, ci risulterà molto utile per le fasi di validazione e di generazione del codice, durante le quali lo si visiterà prendendo in analisi i vari nodi.

Oltre al modello, EMF si occupa anche della creazione di un meta-modello generalmente dedotto dalla grammatica, che identifica i principali concetti e

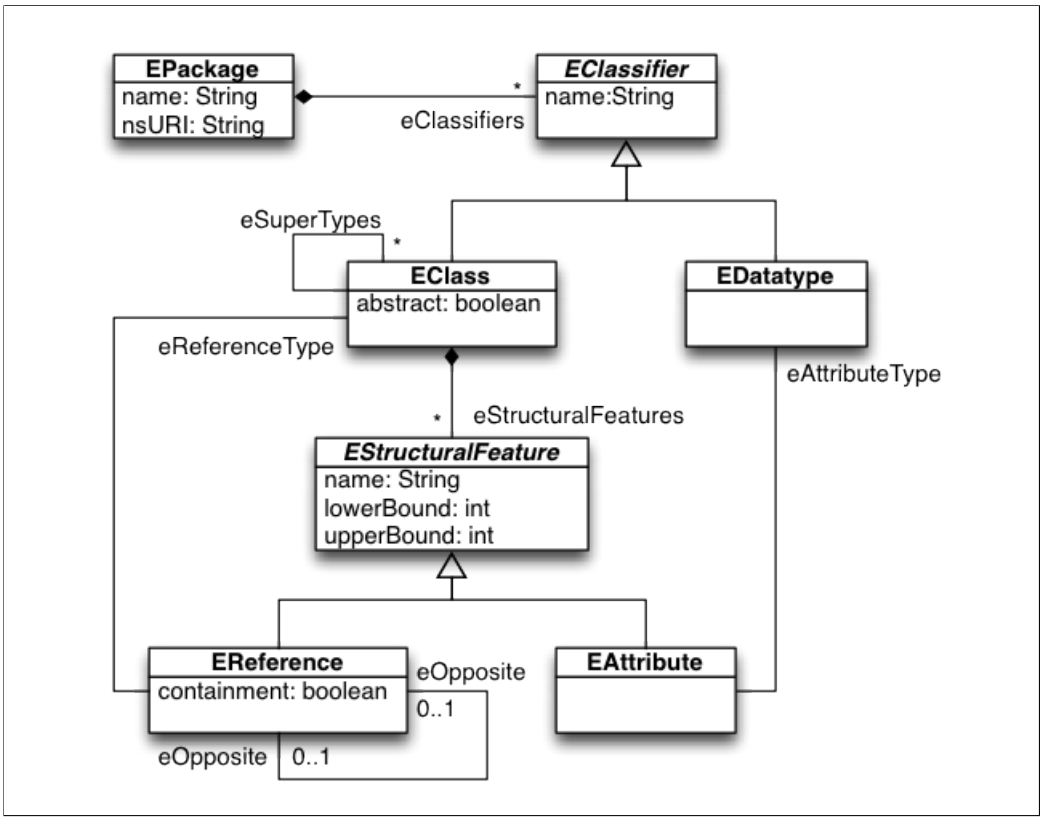

Figura 2.3: I principali concetti di Ecore.

le relazioni tra essi. Il linguaggio in cui il meta-modello è definito prende il nome di *Ecore* e si compone di alcuni concetti fondamentali come mostrato in figura 2.3. Ogni nodo semantico è definito dal meta-modello come una classe UML chiamata `EClass` ed ogni `EClass` può avere una serie di attributi `EAttribute` e una serie di riferimenti `EReference`. Il tipo di ogni `EAttribute` è invece definito in termini di `EDataType`. Alcuni di essi, soprattutto quelli che modellano i relativi tipi primitivi Java come ad esempio `Eint` o `EString`, sono tipi predefiniti di `Ecore`.

Per modellare invece associazioni tra `EClass` si utilizza il concetto di `EReference` che si suddivide sostanzialmente in riferimento di contenimento e in cross-reference (si veda il capitolo 2.1.2). Al fine di rispecchiare la struttura ad albero del modello è imposto il vincolo che ogni `EClass` può avere al più un contenitore.

Un'altra caratteristica importante di EMF ed ampiamente utilizzata in *Xtext* è quella di generare codice Java a partire proprio dal modello Ecore. Il risultato della generazione sarà un'interfaccia Java per ogni `EClass` ed una relativa classe per l'implementazione. Alcuni metodi come ad esempio i *getter* e i *setter* oppure i metodi per risalire al contenitore e al relativo riferimento vengono automaticamente inclusi nell'implementazione di ogni classe.

2.2.2 Scoping

Abbiamo parlato in precedenza della funzionalità offertaci da *Xtext* per utilizzare i riferimenti, ora l'unica cosa che rimane è personalizzare il comportamento in modo da poter tener conto della struttura a blocchi del linguaggio. Ci si potrebbe infatti chiedere qual è la visibilità di una certa variabile nel programma. É proprio questo quello di cui ci si occupa nell'implementazione dello *scope*.

Uno scope viene implementato da *Xtext* come un contenitore che contiene tutti gli oggetti che possono essere raggiunti da un certo riferimento ed ha generalmente una struttura ad albero. La classe utilizzata per la costruzione dello scope prende il nome di *Scope Provider* (generalmente la classe viene chiamata `NomelinguaggioScopeProvider`, nel nostro caso `FallScopeProvider`) e può essere utilizzata nel modo "tradizionale" ossia sfruttando i metodi messi a disposizione come ad esempio il metodo `getScope()`, oppure sfruttando una sintassi dichiarativa dei metodi del tipo `scope_<RefDeclaringEClass>_<Reference>()`. Secondo quest'ultimo approccio, quando bisogna risolvere un riferimento viene cercato e chiamato un metodo che ha la suddetta firma [5].

La funzione `getScope()` prende come parametri un oggetto di tipo `EObject` chiamato *context* e un oggetto di tipo `EReference`. Al suo interno si crea una lista di oggetti visibili in quel contesto e la si passa come parametro alla funzione `scopeFor()` della classe `Scopes`. Questa funzione ha il compito di creare l'albero che sarà poi lo scope finale. Infatti, oltre alla lista di oggetti, è possibile passare un altro parametro che sarà il *parent scope*. Se non viene specificato il secondo parametro, il parent scope sarà settato all'oggetto `IScope.NULLSCOPE`, uno speciale scope nullo.

Gli approcci da utilizzare per la creazione dello scope possono essere differenti, partendo dal riferimento e procedendo a ritroso oppure dalla radice del modello fino al riferimento.

Quello a cui ci si è limitato in questo lavoro di tesi è l'implementazione dello scope interno ossia relativo solamente al nostro programma, non avendo infatti inserito la possibilità di fare riferimenti "esterni" a file o ad altri programmi.

2.2.3 Validazione

La fase di validazione si divide in validazione automatica e custom. La prima riguarda quello che viene implicitamente effettuato dal parser ossia la correttezza sintattica relativamente alla grammatica, oppure il controllo della presenza di cosiddetti *broken references* (riferimenti ad oggetti non ancora istanziati). La seconda invece si occupa di tutti quei vincoli che la grammatica non può coprire. Si pensi ad esempio alla dichiarazione di funzioni ed in particolare ai parametri formali. Si suppone infatti che l'uso di una funzione precedentemente dichiarata tenga conto dell'uguaglianza del numero dei parametri formali e attuali e della compatibilità del loro tipo.

Altri esempi possono essere il controllo di dichiarazioni di variabili che hanno lo stesso nome, oppure la verifica della condizione del costrutto *while*.

Tutto questo viene eseguito all'interno della classe `FallJavaValidator` mediante apposite funzioni precedute dalla clausola `@Check`.

Prendiamo come esempio la funzione seguente in pseudocodice Java:

```
1 @Check
2 private void checkParametersCount(Function f) {
3    Reference ref = f.getReference();
4    if (f.getParameters.size() != ref.getParameters.size())
5       error("Actual and formal parameters count does not match.");
6 }
```

Il funzionamento è molto semplice: la clausola `@Check` dice di chiamare la funzione `checkParametersCount()` ogni volta che un oggetto del tipo indicato come parametro formale viene istanziato (in questo caso il tipo è `Function`). Quello che poi viene fatto nel codice è produrre un errore nel caso in cui il numero di parametri attuali sia diverso del numero di parametri formali della funzione dichiarata (il cui riferimento viene risolto grazie all'implementazione precedente dello scope).

Prendiamo come secondo esempio il controllo di variabili duplicate (in questo esempio non terremo conto della struttura a blocchi e quindi della coesistenza di variabili con lo stesso nome sotto determinate condizioni). In questo caso tutto l'albero dovrà essere visitato e per ogni oggetto `Variable`, esempio di nome per l'`EClass` di una variabile, se ne confronterà il nome con quello delle precedenti variabili.

Supponendo che `Root` sia l'`EClass` della entry-rule, lo pseudo codice corrispondente e non molto efficiente in realtà, sarà simile al seguente:

```
@Check
private void checkDuplicates(Root root) {
   List<Variable> vars = getAllContentsOfType(root, Variable.class);
   for (Variable var : vars) {
      List<Variable> remainingVars = vars.subList(1, vars.size());

      if (remainingVars.contains(var.name()) {
         error("Duplicate variable named '" + var.name() + "'.");
      }
   }
}
```

La funzione in pratica raccoglierà tutti gli oggetti di tipo `Variable` presenti nel programma, a partire dalla radice `Root` e ne confronterà i nomi. In caso di duplicati verrà generato un errore.

In combinazione con la validazione può essere utilizzato il semplice framework *JUnit* per effettuare controlli specifici ed automatizzati. Prima di procedere alla successiva fase occorre ricordare che la validazione è di fondamentale importanza in quanto preparatoria alla generazione del codice. Tanti più controlli saranno effettuati, tanto più sarà facile la traduzione, tenendo conto del fatto che un errore nella generazione del codice corrisponde ad un non funzionamento del programma.

2.2.4 Generazione del codice

La semantica dei costrutti che fanno parte del nostro linguaggio è specificata tramite generazione di codice Java. La scelta di generare codice Java e non direttamente *byte code* vede come motivazione principale quella di rendere disponibile lo sviluppo anche su piattaforme non-JVM, come ad esempio *Google Web Toolkit* (*GWT*) oppure *Android*, ottenendo così un ulteriore livello di astrazione. La classe messa a disposizione da *Xtext* prende il nome, nel caso del nostro linguaggio, di `FallGenerator` ed è generalmente contenuta in un file con estensione *.xtend*.

Xtend è un linguaggio general purpose implementato proprio tramite *Xtext*, con lo scopo di modernizzare Java, semplificando la sintassi per raggiungere una miglior leggibilità ed aggiungendo funzionalità per migliorare alcuni aspetti del linguaggio [6]. Il nome del metodo di partenza per la generazione del codice è `doGenerate()`. Al suo interno tipicamente si visita come meglio si crede l'albero e si traduce ogni nodo.

```
override void doGenerate(Resource resource, IFileSystemAccess fsa) {

   val model = resource.allContents.toIterable
      .filter(typeof(Model)).iterator.next;

   for(stud : model.students)
         fsa.generateFile("Student" + stud.id + ".java",
             compile(stud))

}

def compile(Student student) '''

   private class Student«student.id» {

      private int id = «student.id»;

      private String firstName = "«student.firstName»";

      private String lastName = "«student.lastName»";

      private String phone = "«student.phone.compile»";

      /* COSTRUTTORE */
      public Student«student.id»(int id, String firstName, String
          lastName, String phone) {
         this.id = id;
         this.firstName = firstName;
         this.lastName = lastName;
         this.phone = phone;
      }

   }
'''

def compile(PhoneNumber phone) '''
   «phone.internationalPrefix + phone.nationalPrefix + phone.number
»'''
```

Listato 2.6: Generazione delle classi `Student`.

Facciamo un esempio, supponendo di avere una parte di grammatica del tipo:

```
Model:
   students+=Student+
;

Student:
   id=INT';'
   firstName=STRING';'
   lastName=STRING';'
   phone=PhoneNumber';'
;

PhoneNumber:
   internationalPrefix='+39' nationalPrefix=INT number=INT
;
```

Si vuole trasformare ogni istanza della `EClass Student` in una classe Java, ovviamente su file separati. Il listato 2.6 contiene un esempio di come se ne può generare il codice.

Il tipo di ritorno implicito delle funzioni `compile` è `String` e tutto ciò che si trova fra le doppie parentesi angolari è trattato come la rappresentazione di tale oggetto in base alla funzione `toString()`. Non è difficile quindi capire il semplice codice generato a partire dall'esempio precedente.

2.2.5 Funzionalità aggiuntive

Come si è detto in precedenza, *Xtext* mette a disposizione altre funzionalità con le quali personalizzare ulteriormente il nostro linguaggio. Tra le tante, si ha la possibilità di modificare il comportamento del *content-assist* di Eclipse che di default è "guidato" dalla grammatica ed in particolare dalle scelte che possono essere effettuate dal parser.

Un'altra caratteristica aggiuntiva è la possibilità di modificare la gestione dei cosiddetti *quick-fix*. Eclipse permette infatti, in presenza di warning o errori, di aprire un menu a tendina riportante un eventuale messaggio e le possibili correzioni per sistemare ciò che non va. Ovviamente tali correzioni saranno relative al linguaggio implementato e possono essere di grande aiuto al programmatore.

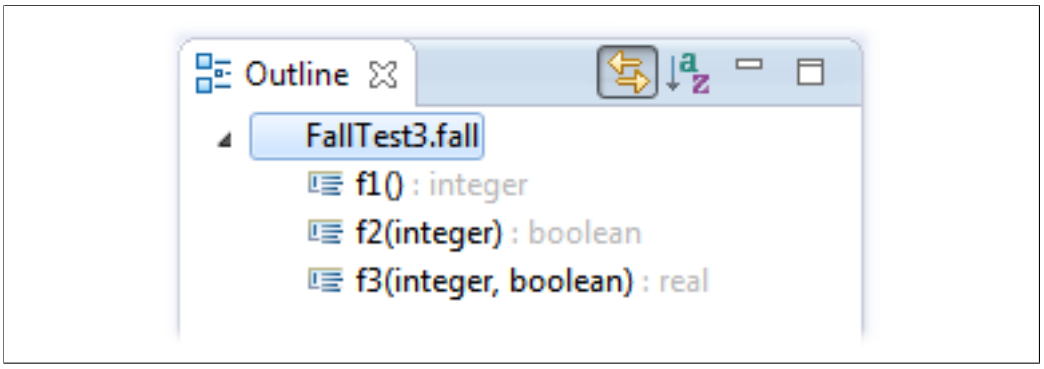

Figura 2.4: Vista della outline.

É inoltre possibile effettuare modifiche riguardanti il *code folding*, ossia quella funzionalità che permette di contrarre o espandere a piacimento porzioni di codice (si pensi ad esempio di voler nascondere il codice riguardante il corpo di una funzione).

Infine ci si può cimentare nella personalizzazione del codice per evidenziare ad esempio parole chiave o stringhe con differenti colori, oppure si può implementare un *formatter* la cui funzione è quella di creare un nuovo flusso di token dopo aver gestito spazi bianchi, a capo o commenti. Il risultato sarà quindi una formattazione del codice corretta e di facile comprensione.

Per questioni di tempistica, molti di questi aspetti come quelli relativi a *content-assist* e ai *quick-fix* sono stati tralasciati. Ciò che è invece stato ritenuto importante implementare è la parte riguardante la *outline* che altro non è che una rappresentazione dell'albero relativo al nostro programma. Di default *Xtext* visualizza tutto l'albero, etichettando ogni nodo con il nome della relativa `EClass`. Si è scelto di rendere il comportamento della outline simile a quello che già viene utilizzato in Java, e che permette di avere una visione dell'interfaccia del programma, ossia le cosiddette *signature* dei metodi che si vanno a dichiarare. Questo lavoro viene svolto dalla classe `MyDslOutlineTreeProvider` in combinazione con un'altra classe `MyDslLabelProvider`. La prima viene utilizzata per scegliere cosa andrà nella outline, la seconda invece è come ci andrà, ossia quale sarà l'etichetta che andrà data ad un tale nodo. In figura 2.4 viene presentata una rappresentazione della outline in stile Java, raffigurante i metodi dichiarati con relativi tipi dei parametri e il tipo del valore di ritorno indicato dopo il simbolo ':'.

Essendo basato su Java, *Xtext* offre la possibilità di integrarsi con tale linguaggio ed in particolare di sfruttarne alcune caratteristiche fondamentali

come ad esempio le espressioni. Si arriverà infatti ad un punto dello sviluppo del proprio linguaggio in cui bisognerà definire come saranno strutturate le espressioni, in modo da integrarle nei suoi costrutti.

Purtroppo la fase di implementazione è complicata non solo dal punto di vista della grammatica (bisognerà ad esempio eliminare le ambiguità e tenere conto della precedenza tra gli operatori), ma soprattutto dalle fasi successive come per esempio quella di validazione in cui bisognerà tenere conto del tipo di una espressione e quella di generazione del codice.

La maggior parte di questi aspetti viene semplificato da *Xbase*, un linguaggio di programmazione "parziale", ossia orientato esclusivamente alle espressioni, implementato con *Xtext* [7].

Google Guice [8] è invece uno strumento utilizzato da *Xtext* con lo scopo di utilizzare funzionalità messe a disposizione da un componente all'interno di un altro componente. Si tratta in pratica di un sistema di *dependency injection* in cui si definisce la dipendenza invece di risolverla permettendo un maggiore riuso del codice così come una migliore manutenibilità del sistema. Questa operazione si effettua in *Xtext* tramite l'uso dell'annotazione Java `@Inject` prima del componente di cui si vuole fare uso.

Supponendo di voler utilizzare il validator nel nostro generatore di codice si avrà qualcosa del tipo:

```
1 class MyGenerator implements IGenerator {
2
3   @Inject
4   private MyValidator validator;
5
6 }
```

Ovviamente Guice dovrà sapere come istanziare oggetti per una determinata dipendenza. Questo viene specificato nei cosiddetti moduli, generalmente di due tipi e creati da *Xtext* per ogni linguaggio. Si tratta dei moduli `MyDslRuntimeModule` e `MyDslUiModule`. Il primo è utilizzato quando il linguaggio viene eseguito al di fuori dall'ambiente Eclipse, il secondo invece per un utilizzo interno ad Eclipse e mette a disposizione principalmente metodi per l'IDE del linguaggio.

Capitolo 3

Il linguaggio FALL1

Questo capitolo vuole dare una panoramica del linguaggio FALL1. Verranno presi in considerazione diversi argomenti, a partire dalla sintassi passando per la spiegazione dei costrutti del linguaggio e dei metodi utilizzati nella fase di validazione fino ad arrivare alla parte relativa alla generazione di codice Java. Per completare il tutto verrà spiegata l'implementazione dello scope oltre ad aspetti secondari quali il *labeling* e la *outline*.

3.1 Sintassi

La sintassi del linguaggio FALL1 è specificata tramite la grammatica riportata nei listati 3.1, 3.2 e 3.3 in una forma semplificata per essere più facile da comprendere. Nel nostro caso si adottano le seguenti convenzioni:

- Ogni elemento compreso fra parentesi angolari è un simbolo non terminale.
- Ogni elemento compreso fra apici oppure totalmente in maiuscolo è un simbolo terminale.
- `x`+ denota una o più occorrenze di `x`.
- `x`* denota zero o più occorrenze di `x`.
- `x`? denota zero o una occorrenza di `x`.
- (`x` | `y`) significa uno tra `x` e `y`.

A causa di un problema nel codice riguardante la struttura a blocchi descritta nel paragrafo 3.1.1, ogni programma scritto in FALL1 deve terminare con una linea vuota per evitare l'insorgere di un errore di lettura dell'ultimo

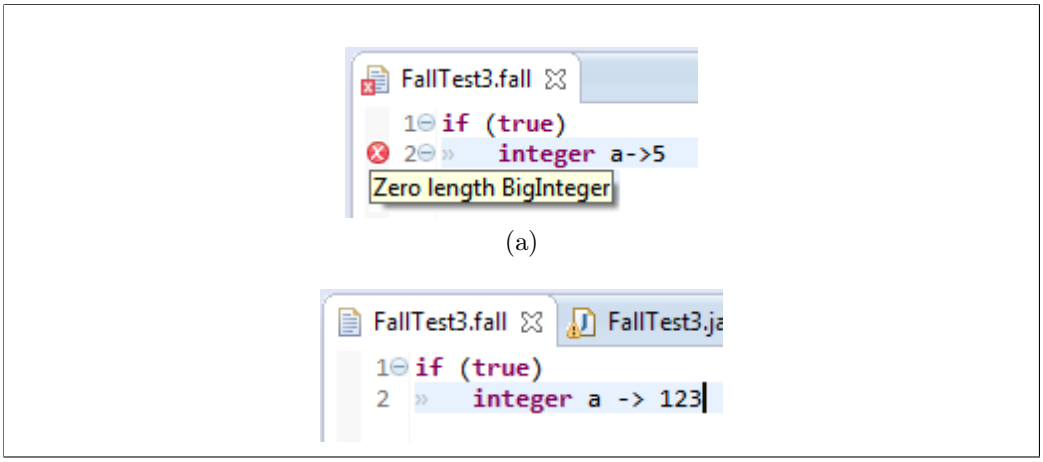

Figura 3.1: Esempi di errori del modulo di indentazione.

carattere. Potrebbero infatti senza tale accorgimento verificarsi situazioni indesiderate come quella mostrata in figura 3.1a. In questo caso il messaggio di errore visualizzato dall'editor è "Zero length BigInteger" ossia è come se la cifra '5' non fosse stata letta. Proprio perché l'editor si è occupato di mostrare un errore, non sarà possibile passare alla fase di generazione del codice. Nella situazione in figura 3.1b si ha invece un comportamento simile ma più pericoloso. Alla variabile `a` è stato assegnato il valore '123' e nessun errore è stato mostrato dall'editor. Questo significa che il processo è stato portato avanti fino alla generazione del codice. Quello che però è stato generato è:

```
1 if ( true ) {
2    BigInteger a = new BigInteger("12");
3 }
```

Come si può notare il valore realmente assegnato alla variabile `a` è '12' e non '123' come ci si aspettava. Questo problema si può anche presentare sotto forma di errore di *highlighting* da parte dell'editor. L'inserimento di una riga vuota al termine del programma permette di evitare questo tipo di errore da parte del modulo per l'indentazione.

```
<main_block> ::=
   <function_definition>* <statement>+

<function_definition>:
   <type> ID '(' (<type> <variable> ( ',' <type> <variable> )* )?
      ')' <block_statement>

<statement> ::=
     <var_declaration>
   | <function_call>
   | <assignment>
   | <if_else>
   | <return_statement>
   | <block_statement >
   | <while_statement>
   | <unification>
   | <for_each>
   | <print_statement>

<block_statement> ::=
   INDENT <statement>+ DEDENT

<print_statement> ::=
     'print' '(' <expression> ')' | 'println' '(' <expression> ')'

<unification> ::=
   '[' <variable> '|' <variable> ']' '=' <variable>

<assignment> ::=
   ( <variable> ('[' <expression> ']')? ) '->' ( <expression> |
      <assignment> )

<function_call> ::=
   ID '(' ( <expression> ( ',' <expression> )* )? ')'

<return_statement>:
   'return' <expression>

<var_declaration> ::=
     <type> <variable> '->' <expression>
   | 'var' <variable> '->' <expression>
```

Listato 3.1: La grammatica del linguaggio FALL1.

```
<expression> ::=
     <expression> '+' <expression>
   | <expression> '-' <expression>
   | <expression> '*' <expression>
   | <expression> '/' <expression>
   | <expression> '=' <expression>
   | <expression> '!=' <expression>
   | <expression> '<' <expression>
   | <expression> '>' <expression>
   | <expression> '<=' <expression>
   | <expression> '>=' <expression>
   | '+' <expression>
   | '-' <expression>
   | '(' <expression> ')'
   | <expression> 'or' <expression>
   | <expression> 'and' <expression>
   | '!' <expression>
   | <expression> 'in' <expression>
   | '(' <type> ')' <expression>
   | '|' <variable> '|'
   | <variable> '[' <expression> ']'
   | <function_call>
   | <variable>
   | <primitive_literal>

<primitive_literal> ::=
     BIGINT
   | REAL
   | STRING
   | 'true'
   | 'false'
   | <list>
   | <set>

<type> ::=
     'integer' | 'real' | 'boolean' | 'string'
   | 'list of integer' | 'list of real' | 'list of boolean'
   | 'list of string'
   | 'set of integer' | 'set of real' | 'set of boolean'
   | 'set of string'
```

Listato 3.2: La grammatica del linguaggio FALL1.

```
<list> ::=
     '[' ']'
   | '[' <expression> ( ',' <expression> )* ']'

<set> ::=
     '{' '}'
   | '{' <expression> ( ',' <expression> )* '}'

<for_each> ::=
   'for' <variable> 'in' <expression> <block_statement>

<while_statement> ::=
   'while' '(' <expression> ')' <block_statement>

<if_else> ::=
   'if' '(' <expression> ')' <block_statement> ( 'else'
      <block_statement> )?

<variable> ::=
   ID
```

Listato 3.3: La grammatica del linguaggio FALL1.

3.1.1 Struttura a blocchi e indentazione

La fase relativa alla scrittura dell'indentazione è stata parecchio problematica, non tanto relativamente al problema in sè, ma quanto all'effettiva realizzazione.

Nel linguaggio FALL1 si è voluto mantenere una struttura a blocchi "visuale" ossia messa in risalto dalla particolare scrittura del codice. É stato quindi scelto un approccio simile a quello già utilizzato nel linguaggio Python secondo cui i blocchi sono definiti dalla loro indentazione. L'idea era quella di aprire un blocco tramite una *newline* seguita da un *tab* e di chiuderlo riportandosi allo stesso livello di indentazione iniziale.

Prendiamo come esempio un blocco etichettato con A che contiene due blocchi, B e E. Il blocco B a sua volta contiene due blocchi C e D. Per ottenere questo annidamento si dovrà produrre un'indentazione simile a quella in figura 3.2.

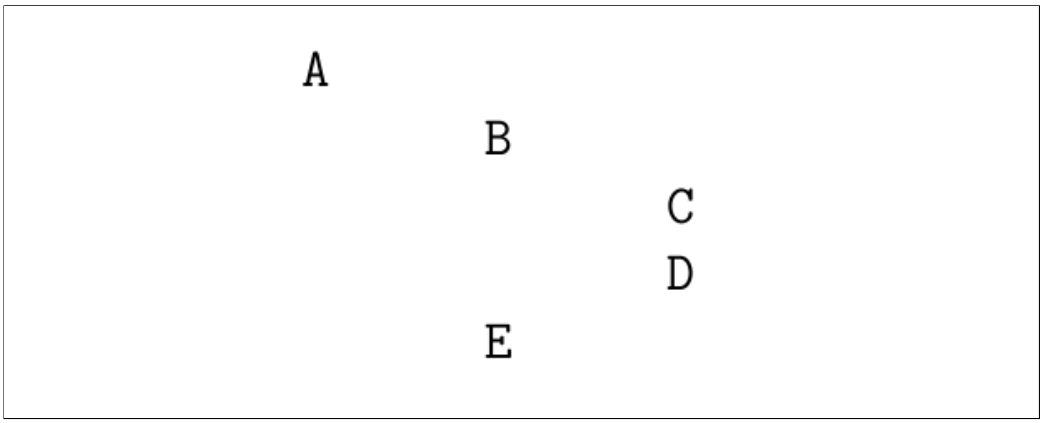

Figura 3.2: Un esempio di blocchi annidati con indentazione.

Per fare ciò è stato utilizzata inizialmente la classe messa a disposizione direttamente da *Xtext* denominata `AbstractSplittingTokenSource`. La sua funzione è quella di suddividere un dato token in token diversi e l'idea era proprio quella di sfruttare questo meccanismo per generare token speciali *INDENT* e *DEDENT* come delimitatori rispettivamente di inizio e fine blocco. A tale scopo sono stati riservati il simbolo '@' e il simbolo '#' nella grammatica per definire le due regole terminali.

L'idea è che, ogni volta che viene incontrata una newline seguita da un tab, la classe `AbstractSplittingTokenSource` si occupa di gestirli generando al loro posto un token INDENT. Inoltre è necessario tenere traccia del livello di annidamento a cui ci si trova per poter generare correttamente anche i token DEDENT relativi alla chiusura di ogni blocco precedentemente aperto.

I due principali metodi messi a disposizione sono:

- `shouldSplitToken()`, che ritorna un booleano e all'interno del quale bisogna specificare quali tipi di token saranno presi in considerazione.

- `doSplitToken()`, ossia il metodo che effettivamente svolge il lavoro di *splitting* sul token precedentemente etichettato come "da processare".

L'approccio visto in questi termini sarebbe stato sicuramente una buona scelta ma, già dai primi test, si è dovuto fare i conti con problemi dovuti probabilmente ad uno sviluppo ancora acerbo della classe stessa.

Nonostante la funzionalità di indentazione fosse stata implementata correttamente, lo *splitting* ha interferito con le normali procedure dell'IDE, risultando in una finestra di errore del *reconciler* di Eclipse ogni qualvolta si

tentava di aprire un blocco [9]. Tutto ciò perché ogni token ha determinate caratteristiche, come ad esempio il punto di inizio e di fine nello *stream* oppure la posizione in cui si trova nella linea, che vengono successivamente processate automaticamente per funzionalità specifiche di Eclipse. Tali caratteristiche, avendo rimaneggiato lo stream, devono essere consistenti, ad esempio il punto di inizio di un token non deve sovrapporsi con il punto di fine del precedente [10]. Pur essendo un concetto semplice, la conseguenza è che tutti i token successivi al primo *splitting* devono essere processati dai metodi della classe `AbstractSplittingTokenSource` per sistemare gli attributi.

Nonostante in seguito a questa correzione sia stato risolto l'errore del *reconciler*, si è comunque andati ad interferire con altre funzionalità di Eclipse. L'editor e la outline ad esempio prendevano ancora in considerazione il modello prima che avvenisse lo *splitting*, con conseguenti errori nella *syntax highlighting* e nella rappresentazione grafica del modello stesso.

A causa di questi problemi e rendendosi conto che la `AbstractSplittingTokenSource` opera su token e non sui singoli simboli (caratteri) in input, si è scelto di cambiare approccio e di modificare direttamente lo stream di caratteri, ossia il *char stream*. In particolare è stata presa una soluzione trovata sul web ed adattata per il nostro scopo.

Quello che si ha a questo punto è sostanzialmente una classe `Tokenizer` che si occupa di raccogliere e gestire i token di INDENT e DEDENT ed una classe `IndentingStream` che implementa `CharStream` e sfrutta il tokenizer per tenere conto dei token di indentazione. Seguendo questo schema non si è più vincolati ad avere necessariamente blocchi dettati dal numero di tab, ma si ha una vera e propria struttura *whitespace-aware*. Ciò significa che per segnalare l'inizio di un blocco possono essere utilizzati anche i singoli caratteri di spaziatura e non necessariamente il tabulatore.

Il passo successivo è stato creare un lexer che tenesse conto del nuovo char stream e successivamente creare un parser che sfruttasse invece il lexer precedente. A questo punto l'unica cosa rimasta è stata effettuare il *binding* del nuovo parser nel modulo di runtime chiamato `FallRuntimeModule` di *Xtext* attraverso l'overriding della funzione `bindIParser()`.

A seguito di quest'ultima soluzione, la quasi totalità dei problemi riguardanti l'indentazione è stata risolta.

3.2 Principali costrutti

In questo paragrafo verranno illustrati in linea generale i principali costrutti appartenenti al linguaggio FALL1 e verranno forniti alcuni esempi di utilizzo

pratico.

3.2.1 Assegnamento e tipi primitivi semplici

I tipi primitivi semplici presenti nel linguaggio ed indicati da qui in avanti come *sType*, sono quattro:

- `integer` rappresenta interi di grandezza arbitraria con o senza segno. Con grandezza arbitraria si intende che finché si ha a disposizione memoria sul calcolatore, tale memoria può essere utilizzata per allocare il numero intero. Si ha quindi che l'unico limite è quello fisico relativo alla quantità di memoria disponibile.

- `real` include i numeri a virgola mobile di grandezza arbitraria (come detto sopra l'unico limite è dato dalla memoria disponibile) rappresentati come parte intera e parte frazionaria separate da '.', il punto decimale.

- `boolean` assume solamente due valori '*true*' o '*false*'.

- `string` rappresenta una stringa di caratteri.

Oltre ad essi sono presenti due ulteriori tipi primitivi che permettono di dichiarare variabili di tipo lista o di tipo insieme:

- `list of <sType>`

- `set of <sType>`

Per quanto riguarda le operazioni tra liste e insiemi si faccia riferimento al paragrafo 3.2.2.

Definiti i tipi presenti nel linguaggio, l'operatore freccia '`->`' rappresenta un assegnamento. In particolare si può definire una variabile ed assegnare ad essa un valore.

Una dichiarazione di variabile può avere sostanzialmente due forme differenti. La prima è indicata come:

```
<type> <varName> -> <value>
```

Si noti che in questo contesto *type* rappresenta un qualsiasi tipo dato del linguaggio, liste e insiemi inclusi. Inoltre è bene sottolineare che è stato scelto di rendere obbligatoria l'inizializzazione delle variabili tramite un *default value*.

La seconda forma di dichiarazione di una variabile è più generale e sfrutta il meccanismo di controllo dei tipi implementato nel *validator*. Utilizzando quindi il seguente comando:

`var <varName> -> <value>`

si avrà che la variabile di nome `varName` avrà lo stesso tipo dato dalla valutazione del suo `value`. Ad esempio il codice seguente implica che il tipo dato della variabile `a` è `list of string`.

```
var a -> ["red", "green", "blue"]
```

La dichiarazione di una variabile di tipo lista o insieme viene specificata con i due comandi

```
list of <sType> <listName> -> [ <elements> ]
set of <sType> <setName> -> { <elements> }
```

dove `sType` è uno dei tipi primitivi semplici messi a disposizione dal linguaggio ed `elements` è l'insieme, eventualmente vuoto, degli elementi contenuti separati da virgole. Se vogliamo creare una lista di interi e un insieme di stringhe scriveremo ad esempio:

```
list of integer myList -> [1, 2, 3]
set of string mySet -> {"one", "two", "three"}
```

Come nel caso delle dichiarazioni di variabili, l'operatore '`->`' ovviamente permette allo stesso modo di assegnare un valore ad una variabile precedentemente dichiarata. Inoltre è possibile esprimere una catena di assegnamenti del tipo:

```
var_1 -> var_2 -> ... -> var_n -> value
```

Il controllo dei tipi implementato nel *validator* rileverà eventuali errori durante tale assegnamento. Si noti, infine, che nel linguaggio FALL1 un assegnamento non è considerato un'espressione.

3.2.2 Operazioni primitive, tra liste e tra insiemi

Le operazioni tra tipi di dato primitivi semplici sono le tradizionali di addizione, sottrazione, moltiplicazione e divisione per i tipi `integer` e `real` mentre solamente di addizione, ossia la concatenazione, per il tipo `string`.

Oltre a questi operatori vengono messi a disposizione anche quelli di confronto '=', '!=', '<', '>', '<=', '>=' e di negazione '!'. Sono inoltre presenti gli operatori logici di congiunzione 'and' e di disgiunzione 'or'.

In aggiunta alla presenza di operazioni su dati primitivi semplici come gran parte dei moderni linguaggi di programmazione, FALL1 mette a disposizione la possibilità di creare e manipolare con semplici operazioni liste ed insiemi. Le operazioni tra liste ed insiemi sono sostanzialmente le medesime e tengono conto della particolare natura delle due strutture dati. Gli operatori utilizzati sono gli stessi che si utilizzerebbero tra tipi primitivi.

I primi operatori che si vogliono introdurre sono gli operatori di manipolazione '+' e '-'. Entrambi sono operatori binari ed il loro significato è quello rispettivamente di aggiunta e rimozione di elementi. Mentre nessuna chiarificazione è necessaria per quanto riguarda queste operazioni tra insiemi, qualche osservazione va fatta quando si agisce su liste. In questo caso, l'operatore '+' rappresenta in effetti una concatenazione di una lista con un'altra mentre l'operatore '-' rimuove dalla prima lista tutti gli elementi (se esistono) della seconda, duplicati compresi. Il codice seguente ad esempio:

```
1 list of string abc -> ["a", "b", "c"]
2 list of string def -> ["d", "e", "f"]
3
4 println(abc + def)
5
6 var acbc -> ["a", "c", "b", "c"]
7
8 println(acbc - ["c"])
```

visualizza a video l'output:

```
[a, b, c, d, e, f]
[a, b]
```

Un altro operatore messo a disposizione è quello di confronto '=' che si comporta in due modi differenti per liste o insiemi. L'operatore di confronto tra due liste ritorna vero se le due liste hanno la stessa cardinalità (si veda il paragrafo 3.2.3) e gli stessi elementi nel medesimo ordine. Lo stesso operatore applicato a due insiemi restituisce vero se i due insiemi hanno la stessa cardinalità e contengono gli stessi elementi, indipendentemente dal loro ordine. Ad esempio il codice riportato nel listato 3.4 produce come output:

```
1  list of integer oneToThree -> [1, 2, 3]
2  list of integer threeToOne -> [3, 2, 1]
3
4  if (oneToThree = [1, 2, 3])
5     println("true")
6  else
7     println("false")
8
9  if (oneToThree = threeToOne)
10    println("true")
11 else
12    println("false")
```

Listato 3.4: Confronto fra liste.

```
true
false
```

Lo stesso codice opportunamente modificato per operare su insiemi produrrà invece l'output:

```
true
true
```

Per quanto riguarda le liste è inoltre possibile accedere ad un elemento di indice `index` tramite la sintassi:

```
list[index]
```

come avviene ad esempio in Java o C++. Così come in questi due linguaggi, la numerazione degli elementi in FALL1 parte da zero.

3.2.3 Operatore di cardinalità

L'operatore di cardinalità può essere applicato su liste o insiemi ed il suo scopo è quello di ritornare il numero di elementi presenti. La sua sintassi è specificata come segue:

```
| listOrSet |
```

Chiaramente, come ci si aspetta, in una lista vengono tenuti in considerazione anche gli elementi duplicati, cosa che non avviene invece per gli insiemi.

Prendiamo come esempio una lista e un insieme di dieci numeri casuali e vediamo come differisce il risultato dell'operatore di cardinalità:

```
1 list of integer intList -> [0, 2, 2, 4, 5, 5, 6, 7, 9, 9]
2 println("List elements count: " + |intList|)
3 set of integer intSet -> {0, 2, 2, 4, 5, 5, 6, 7, 9, 9}
4 println("Set elements count: " + |intSet|)
```

Ciò che viene stampato a video sarà:

```
List elements count: 10
Set elements count: 7
```

3.2.4 If-else

Il costrutto *if-else* si struttura nel seguente modo:

```
if ( condition )
    statement_1
    statement_2
    ...
    statement_n
else
    statement_n+1
    statement_n+2
    ...
    statement_m
```

ed ha la semantica a tutti nota. Nel paragrafo 2.1.4 è stata descritta la famosa situazione del *dangling else*, problema che non deve neanche essere preso in considerazione nel linguaggio FALL1. Questo perché la presenza dell'indentazione come inizio e fine blocco implica che il ramo `else` appartiene sempre all'`if` che si trova al suo stesso livello.

Come mostrato in figura 3.3a il ramo `else` appartiene all'`if` più esterno, mentre in figura 3.3b appartiene a quello più interno. Nel nostro caso quindi non è nemmeno necessario utilizzare nella grammatica i predicati sintattici messi a disposizione da ANTLR.

```
if ( condition )
    if ( anotherCondition )
        doSomething
else
    doSomethingElse
```

(a)

```
if ( condition )
    if ( anotherCondition )
        doSomething
    else
        doSomethingElse
```

(b)

Figura 3.3: Appartenenza del ramo else a seconda dell'indentazione.

3.2.5 Ciclo while

Così come il costrutto *if-else*, nel linguaggio FALL1 il ciclo while è essenzialmente lo stesso dei moderni linguaggi di programmazione. Sintatticamente un ciclo while è espresso come:

```
while ( condition )
    statement_1
    statement_2
    ...
    statement_n
```

Il suo significato semantico è quello noto a tutti per cui gli n statement del blocco vengono eseguiti in sequenza e ciclicamente fintanto che la condizione specificata viene valutata come vera. Nel momento in cui la condizione viene valutata falsa, si esce dal ciclo.

Ad esempio il codice seguente stamperà a video il numero 55, ossia la somma dei primi dieci numeri.

```
1 integer i -> 1
2 integer sum -> 0
3 while (i <= 10)
4    sum -> sum + i
5    i -> i + 1
6 println(sum)
```

3.2.6 Ciclo for-each

Il ciclo *for-each* è utile per effettuare operazioni su elementi di liste ed insiemi. La sintassi è specificata come segue:

```
for varName in listOrSet
    statement_1
    statement_2
    ...
    statement_m
```

L'idea di base è quella che troviamo anche nel costrutto for-each di Java.

Supponendo che la cardinalità di `listOrSet` sia n, vengono eseguite n iterazioni del ciclo, una per ogni elemento. All'inizio della i-esima iterazione viene assegnato il valore dell'i-esimo elemento alla variabile locale al ciclo `varName`. In questo modo vengono presi in considerazione in modo sequenziale tutti gli elementi di `listOrSet`.

3.2.7 Operatore di unificazione

L'operatore di unificazione opera su liste di elementi ed ha la seguente sintassi:

```
[ head | rest ] = target
```

Data una lista `target`, l'operatore di unificazione estrae la testa della lista mettendola nella variabile specificata da `head`, mentre ciò che rimane verrà inserito nella variabile specificata da `rest`.

Si noti che `head` è una variabile "semplice", ossia non una lista o un insieme, e deve avere tipo compatibile con quello di `target`. Inoltre `rest` è a sua volta una lista ed anche il suo tipo deve essere compatibile con quello di `target`. Ad esempio il seguente codice produrrà un errore di tipo poiché si sta cercando di assegnare la testa di `catDogMouse`, ossia una stringa, alla variabile `cat` che è un intero.

```
1 list of string catDogMouse -> ["cat", "dog", "mouse"]
2 integer cat -> 0
3 list of string dogMouse -> []
4 [cat | dogMouse] = catDogMouse
```

3.2.8 Espressioni

Nel linguaggio FALL1 le espressioni sono state implementate da zero, senza ricorrere all'aiuto dello strumento *Xbase*. Per fare ciò sono state utilizzate diverse tecniche combinate insieme, alcune delle quali sono state mostrate nel paragrafo 2.1.3.

Oltre alla struttura delle espressioni si è dovuto tenere conto della precedenza dei vari operatori. Come ben spiegato infatti nell'articolo di Sven Efftinge (capo progetto di *Xtend* e *Xtext*) la precedenza degli operatori è stabilita dalla catena di delegazione delle regole, ossia più tardi una regola è chiamata, più alta è la sua precedenza [11].

Nel linguaggio FALL1 si avrà il seguente ordinamento dall'operatore con precedenza più alta a quello con precedenza più bassa:

- parentesi tonde
- cast esplicito
- operatori unari '`!`', '`-`' e '`+`'
- espressione di contenimento
- moltiplicazione e divisione
- addizione e sottrazione
- operatori binari '`<`', '`<=`', '`>`' e '`>=`'
- operatori binari '`=`' e '`!=`'
- operatore logico '`and`'
- operatore logico '`or`'

Sebbene il significato dei suddetti costrutti sia abbastanza chiaro, ci si vuole soffermare sull'espressione di contenimento. Tale espressione ha la seguente forma:

```
lExp in rExp
```

Il suo tipo risultante è booleano ed ha senso se utilizzata nel contesto di liste ed insiemi. Il suo significato semantico infatti è di controllare se l'elemento dato dalla valutazione di `lExp` è contenuto nella lista o nell'insieme dato dalla valutazione di `rExp` (la correttezza del tipo di quest'ultima espressione è determinata dalla funzione `checkContainmentExpressions()` descritta nel paragrafo 3.4).

Una volta strutturate le espressioni rimane da implementare il *type-checking*, ossia il controllo che un dato valore sia effettivamente di un tipo compatibile con la variabile alla quale verrà assegnato. Il type-checking è eseguito all'interno del *validator* e si basa sulla mappatura da parte del metodo `checkType()` di un tipo ad ogni espressione. Tale metodo prende in input un'espressione e restituisce un tipo definito nel costrutto *enum* chiamato `Type` (si veda il listato 3.5).

In particolare il tipo `Type.INVALID` sta ad indicare che una determinata espressione non ha un tipo corretto, come ci si aspetta che avvenga ad esempio sommando un `integer` con un `boolean`. A seconda del tipo degli operandi esistono quindi operazioni valide oppure invalide.

La lista delle operazioni consentite è stata inserita in due `HashMap` [12, p. 388], la prima contenente quelle binarie e che mappa una *tripla*

$$\langle\ lType,\ operand,\ rType\ \rangle$$

in un `Type`, mentre la seconda racchiude le operazioni unarie mappando una coppia

$$\langle\ operand,\ Type\ \rangle$$

nel suo `Type` risultante. Prendiamo come esempio l'operazione binaria di disuguaglianza e consideriamo il codice appartenente all'implementazione di `checkType()`:

```
1 map.put(new Triplet<>(Type.INTEGER, "!=", Type.INTEGER),
      Type.BOOLEAN);
```

Il suo significato può essere spiegato a parole nel seguente modo: se l'espressione di sinistra e quella di destra valutano al tipo `Type.INTEGER` e l'operatore è quello di disuguaglianza, allora il tipo risultante dell'espressione binaria è `Type.BOOLEAN`. Il tipo di liste e insiemi non vuoti è determinato dal tipo del loro primo elemento, ad esempio i tipi delle espressioni seguenti

```
1 [1, 2, 3.0]
2 {3.14, 2}
3 [0, "zero"]
```

```
public enum Type {

      INTEGER,
      REAL,
      STRING,
      BOOLEAN,
      LIST_OF_INTEGER,
      LIST_OF_REAL,
      LIST_OF_STRING,
      LIST_OF_BOOLEAN,
      SET_OF_INTEGER,
      SET_OF_REAL,
      SET_OF_STRING,
      SET_OF_BOOLEAN,
      LIST_EMPTY,
      SET_EMPTY,
      INVALID

}
```

Listato 3.5: Tipi delle espressioni.

saranno rispettivamente `Type.LIST_OF_INTEGER`, `Type.SET_OF_REAL` ed infine `Type.INVALID`. Nel primo e nel terzo caso però il validator produrrà un errore a tempo di compilazione. Per quanto riguarda la lista `[0, zero]` si ha un tipo invalido perché, come ci si aspetta, una stringa non può essere convertita in un intero (a meno che tale stringa non contenga un numero intero, ad esempio la stringa "2"). Nella lista `[1, 2, 3.0]` si ha invece un vincolo espresso nel paragrafo 3.4 che impedisce la conversione da `real` a `integer` a causa di una possibile perdita di precisione. La soluzione in questo caso può essere quella di riscrivere la lista come `[1, 2, (integer)3.0]` eseguendo così un cast esplicito.

3.2.9 Stampa a video

Per visualizzare qualcosa a video sono stati utilizzati due costrutti simili `print()` e `println()`. Come nei moderni linguaggi di programmazione il

secondo differisce dal primo per il fatto che stampa come carattere finale il simbolo di *new line*, ossia un a capo.

3.3 Scope

Come già accennato in precedenza nel linguaggio FALL1 si è dovuto sovrascrivere il comportamento di default relativo allo scope, ovvero la risoluzione dei riferimenti. L'implementazione si trova nella classe `FallScopeProvider` che estende `AbstractDeclarativeScopeProvider`. In questo metodo viene distinto lo scope per la risoluzione dei riferimenti delle chiamate a funzioni (precedentemente definite) e lo scope relativo alle variabili nei blocchi.

Nel primo caso raccogliere i nomi delle funzioni definite è molto semplice poiché si trovano in un punto preciso del programma, ossia all'inizio, e la loro dichiarazione è lineare. Nel secondo caso invece, a causa della struttura a blocchi del linguaggio, è stata pensata un'implementazione differente. Siccome la funzione `getScope()` viene chiamata ogni volta che si ha un riferimento, si è scelto un approccio di tipo ricorsivo, utilizzando un metodo ausiliario di nome `buildScope()`. Al suo interno, a partire dal riferimento ancora non risolto, vengono raccolte tramite la funzione `getVarDecarations()` tutte le dichiarazioni di variabili appartenenti allo stesso blocco e che precedono lo *statement* in cui compare il riferimento. Il corpo del metodo `buildScope()` è riportato nel listato 3.6.

Per facilitare la raccolta è stato utilizzato il metodo parametrico in `T getContainerOfType()` messo a disposizione dalla classe `EcoreUtil2` che, presi come parametri un `EObject` e un oggetto di tipo `Class<T>`, risale il *model* restituendo un oggetto (*container*) di tipo `T`.

A questo punto viene settato il *parent scope* per lo scope corrente tramite la chiamata ricorsiva a `buildScope()`. Il risultato sarà una catena di chiamate che arriverà fino al blocco principale del programma il cui parent scope verrà settato allo scope nullo.

3.4 Validazione

La fase di validazione, oltre ad essere complementare alla scrittura della grammatica, è di fondamentale importanza. La sintassi permette infatti di definire in modo generale ciò che si vuole scrivere ma in alcuni casi non consente di esprimere determinati vincoli. Ad esempio ci si renderà presto conto che quasi tutti i metodi contengono un certo numero di controlli sui tipi, vincolo non esprimibile solamente attraverso la stesura della grammatica.

```
1  private IScope buildScope(EObject block, EObject context) {
2
3     EObject parentBlock = block.eContainer();
4
5     if (parentBlock instanceof FuncDefinition || parentBlock
          instanceof MainBlock) {
6
7        return Scopes.scopeFor(getVarDeclarations(parentBlock,
             context));
8
9     }
10
11    else if (parentBlock instanceof BlockStatement || parentBlock
          instanceof ForEach) {
12
13       return Scopes.scopeFor(getVarDeclarations(parentBlock,
             context), buildScope(parentBlock, context));
14
15    }
16
17    else {
18
19       return buildScope(parentBlock, context);
20
21    }
22
23 }
```

Listato 3.6: Il corpo del metodo buildScope().

Di seguito è riportata una descrizione dei metodi di *check* utilizzati all'interno del validator. Tutti i metodi hanno come tipo di ritorno `void` e gestiscono al loro interno eventuali messaggi di errore. Inoltre si vuole fare notare che a causa della scelta di utilizzare alcuni dati primitivi a precisione arbitraria, la conversione dal tipo `real` a `integer` non è ammessa a meno che non venga usato un esplicito cast. In questo modo si riducono gli errori di programmazione relativi alla perdita di precisione. Ovviamente la conversione da `integer` a `real` è ammessa anche senza cast non comportando alcun

problema di troncamento.

Quando si ha una chiamata ad una funzione è fondamentale controllare che il numero e il tipo dei parametri attuali rispecchi la *signature* della dichiarazione della stessa funzione. Lo scopo del metodo `checkActualParametersTypeAndCount()` è esattamente questo. In primo luogo viene verificato se vi è o meno il *match* sul numero dei parametri. Se questo test è superato, si passa al controllo dei tipi. Tale controllo sfrutta il metodo `checkType()` descritto nel paragrafo 3.2.8.

Errori generati dal metodo `checkActualParametersTypeAndCount()`:

- Il numero dei parametri attuali non coincide con il numero dei parametri formali.
- Almeno un tipo di un parametro attuale non è compatibile con il tipo del relativo parametro formale.

Con l'inizializzazione (obbligatoria) di una variabile, è necessario verificare il valore ad essa assegnato. Lo scopo del metodo `checkAssignedType()` è quello di verificare al momento della dichiarazione di una variabile che il valore ad essa assegnato sia compatibile con il suo tipo.

Errori generati dal metodo `checkAssignedType()`:

- Il tipo del valore di *default* non è compatibile con il tipo della variabile.
- La variabile non è tipata (si veda la parola chiave `var` nel paragrafo 3.2.1) e il suo valore assegnato è una lista o un insieme vuoto. In questo caso non può esserne dedotto il tipo a meno che non venga esplicitato un *cast*.

In modo simile a come avviene per il precedente, il metodo `checkAssignmentCompatibility()` prende in rassegna tutti gli assegnamenti, invece delle dichiarazioni di variabili. Siccome sono previsti assegnamenti multipli, è stato scelto di implementare il controllo dei tipi tramite una chiamata ricorsiva al metodo stesso.

Errori generati dal metodo `checkAssignmentCompatibility()`:

- Il tipo della parte destra dell'assegnamento non è compatibile con il tipo della parte sinistra.

Nella maggior parte delle espressioni non è stato necessario effettuare particolari verifiche, se non quelle di correttezza di tipo. Al contrario, una espressione di contenimento ha senso solamente se applicata nel contesto di liste o insiemi. Il metodo `checkContainmentExpressions()` verifica che in una espressione del tipo `x in y`, la variabile `y` sia una lista oppure un insieme.

Errori generati dal metodo `checkContainmentExpressions()`:

- La variabile della parte destra non è una lista o un insieme.

Così come avviene per il metodo relativo ad una espressione di contenimento, nel costrutto *for-each*, quando si ha `for a in x`, l'oggetto `x` su cui si vuole iterare deve essere una lista oppure un insieme. Il metodo `checkForEachExpressionType()` ne effettua semplicemente il controllo.

Errori generati dal metodo checkForEachExpressionType():

- L'oggetto su cui si vuole iterare non è una lista o un insieme.

Un'altra espressione su cui sono state effettuati controlli è quella di *size*. Come già detto nel paragrafo 3.2.3, l'operatore di cardinalità opera su un oggetto di tipo lista o insieme. Il metodo `checkSizeExpression()` si occupa di questo controllo.

Errori generati dal metodo checkSizeExpression():

- L'elemento non è una lista o un insieme.

Nel caso del linguaggio FALL1 è stato necessario modificare il comportamento dello *scope provider* in modo da rispecchiare la regola di visibilità del linguaggio Java che non corrisponde alla regola di visibilità "classica". Tale regola infatti dice che una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome. In tal caso, nel blocco in cui compare la ridefinizione la nuova variabile *nasconde* (o maschera) la precedente [13, p. 73]. In Java infatti una scrittura di questo tipo prevede un errore a compile-time:

```
1 { int a = 2;
2   {
3    int a = 3;
4   }
5 }
```

Il risultato è che non sono possibili ridichiarazioni all'interno dei blocchi annidati. Le specifiche di Java SE 7 dicono infatti che lo scope di una dichiarazione locale di variabile in un blocco è il resto del blocco in cui la dichiarazione compare, a cominciare dal suo inizializzatore e incluso ogni ulteriore dichiaratore a destra nello statement di dichiarazione di variabile. É un errore di compilazione se il nome di una variabile locale `v` è ridichiarato come variabile locale del metodo, costruttore o blocco di inizializzazione che lo contiene direttamente all'interno dello scope di `v` [14]. Lo scopo del metodo `checkDuplicates()` è quello di verificare che la visibilità di una variabile sia corretta in accordo con quella del linguaggio Java.

Errori generati dal metodo checkDuplicates():

- Una variabile con lo stesso nome è già stata precedentemente dichiarata.

Come avviene nel precedente metodo, lo stesso concetto deve essere applicato per il ciclo *for-each*. Al suo interno è presente una variabile che non deve essere visibile al di fuori dello scope del costrutto stesso. Nella sintassi `for a in x`, infatti, il nome `a` è in effetti una dichiarazione implicita di variabile (locale al blocco principale). Una dichiarazione con lo stesso nome non dovrà quindi essere presente a ritroso nel codice.

Errori generati dal metodo checkDuplicatesInForEachBlock():

- Una variabile con lo stesso nome è già stata precedentemente dichiarata.
- Una variabile con lo stesso nome della variabile locale al costrutto è dichiarata all'interno del blocco principale.

Come per il caso appena discusso, anche `checkDuplicatesInFunction()` si occupa di gestire variabili con lo stesso nome ma stavolta all'interno della definizione di una funzione. Il nome di un parametro formale deve essere infatti unico in due sensi: non deve comparire come nome di un altro parametro della stessa funzione e non deve nemmeno essere ridefinito all'interno del corpo.

Errori generati dal metodo checkDuplicatesInFunction():

- Almeno due parametri hanno lo stesso nome.
- Una dichiarazione di variabile ha lo stesso nome di un parametro formale.

Considerando sempre le funzioni, il controllo sui nomi si limita ad un comportamento simile a quello visto per le variabili e per ora non prende in

considerazione la *signature* (questa caratteristica può comunque essere sviluppata nelle future versioni del linguaggio), ma si limita a controllare che un'altra funzione con lo stesso nome non sia già stata definita. Inoltre, per necessità esistono funzioni predefinite del linguaggio i cui nomi non devono essere utilizzate dal programmatore.

Errori generati dal metodo checkRedefinedFunction():

- Due funzioni con lo stesso nome sono state definite.
- Il nome di una funzione corrisponde al nome di una funzione predefinita del linguaggio.

Durante un'operazione di unificazione si ha la presenza di tre entità, la testa della lista (*head*), il resto della lista (*rest*) e la lista da separare (*target*). Il metodo `checkUnification()` si occupa di verificare che le condizioni di applicabilità dell'operatore di unificazione siano rispettate. In particolare vengono controllati i tipi di tutti gli operandi e la relativa compatibilità tra essi.

Errori generati dal metodo checkUnification():

- `head` non è una variabile che ha tipo primitivo semplice.
- `rest` non è una lista.
- `target` non è una lista.
- Il tipo di `head` non è compatibile con il tipo di `target`, cioè non si può estrarre un elemento da `target` e metterlo in `head`.
- Il tipo di `rest` non è compatibile con il tipo di `target`.

Quando si ha a che fare con il costrutto *if-else*, la condizione espressa deve assumere i valori `true` o `false` cioè avere un tipo booleano. In particolare nel nostro linguaggio dovrà assumere un `Type.BOOLEAN`. Il metodo `checkIfThenElseConditionType()` si occupa di questa verifica.

Errori generati dal metodo checkIfThenElseConditionType():

- La condizione specificata non ha valore booleano.

Analogamente a quanto avviene per il costrutto *if-else*, nel metodo `checkWhileConditionType()` viene effettuato un controllo di tipo sulla condizione che riguarda il costrutto *while*. Anche l'errore generato è lo stesso.

L'ultimo controllo effettuato è rivolto allo statement di *return* ed è effettuato dal metodo `checkReturn()`. Nel nostro linguaggio non è stato previsto il tipo di ritorno `void` per cui si ha che ogni funzione deve ritornare un valore. Il controllo è effettivamente molto semplice e considera solamente la presenza dello statement `return` oltre alla compatibilità del valore ritornato. In realtà però i controlli che effettua il linguaggio Java che sta alla base di FALL1, sono molti di più. Ad esempio se è presente comunque un `return` ma si trova in una porzione di codice non raggiungibile (*dead code*) verrà segnalato un errore. Esiste per questo motivo una possibilità per cui, nonostante il codice scritto nel linguaggio FALL1 compili, esso non venga eseguito a causa della presenza di errori nel codice generato.

Errori generati dal metodo checkReturn():

- Non è presente uno statement `return`.
- Il valore ritornato non è compatibile con il tipo di ritorno della funzione.

3.5 Traduzione in Java

La fase finale relativa alla generazione del codice è delegata alla classe `FallGenerator` ed in particolare al metodo `doGenerate()`. Al suo interno si è scelto di generare per ogni programma un file (una classe) con nome lo stesso scelto per il sorgente ed estensione *.java*.

Anche in questo caso la generazione del codice sfrutta il modello ossia come già detto in precedenza l'albero di sintassi astratta, partendo dal nodo radice fino ad arrivare alle foglie. Per fare questo è stata definita una funzione denominata `compile()` per ogni costrutto, ossia `EClass` relativa al linguaggio FALL1, di cui si vuole generare il codice. Per alcune operazioni particolari è stato necessario definire metodi ausiliari che potrebbero essere chiamati a supporto della traduzione. Tali metodi prendono il nome di *funzioni predefinite del linguaggio*.

Di seguito verranno elencate alcune di queste funzioni insieme alle varie `compile()` spiegandone in breve il funzionamento. Si ricorda inoltre che il linguaggio utilizzato dal generatore è *Xtend* (paragrafo 2.2.4).

A partire dalla radice del modello viene compilato il `MainBlock`. All'interno del metodo viene generata la classe principale denominata come il secondo parametro che viene passato, insieme agli *import* Java necessari e alle funzioni predefinite del linguaggio. Queste funzioni sono di supporto per molte situazioni, come la creazione di una lista o le operazioni fra tipi dato del

```
def compile(Statement statement) {
   '''«IF statement.statementType instanceof VarDeclaration»
         «(statement.statementType as VarDeclaration).compile»;
      «ELSEIF statement.statementType instanceof FuncCall»
         «(statement.statementType as FuncCall).compile»;
      «ELSEIF statement.statementType instanceof Assignment»
         «(statement.statementType as Assignment).compile»;
      «ELSEIF statement.statementType instanceof PrintStatement»
         «(statement.statementType as PrintStatement).compile»;
      «ELSEIF statement.statementType instanceof ReturnStatement»
         «(statement.statementType as ReturnStatement).compile»;
      «ELSEIF statement.statementType instanceof BlockStatement»
         «(statement.statementType as BlockStatement).compile»
      «ELSEIF statement.statementType instanceof IfThenElse»
         «(statement.statementType as IfThenElse).compile»
      «ELSEIF statement.statementType instanceof While»
         «(statement.statementType as While).compile»
      «ELSEIF statement.statementType instanceof Unification»
         «(statement.statementType as Unification).compile»;
      «ELSEIF statement.statementType instanceof ForEach»
         «(statement.statementType as ForEach).compile»«
      ENDIF»'''
   }
```

Listato 3.7: Dispatch degli statement.

linguaggio fino ad arrivare alle conversioni di tipo. Come prima cosa vengono compilate le definizioni di funzioni, secondariamente ogni statement è tradotto in Java ed inserito all'interno del metodo `main()` in modo che il programma possa essere reso eseguibile. La compilazione di ogni statement è gestito da una specifica `compile()` che prende come parametro uno `Statement` e si occupa del suo *dispatch* al relativo metodo di traduzione in Java, come si può vedere nel listato 3.7. Al suo interno viene verificato che l'attributo `statementType` sia istanza di una `EClass` relativa ad un preciso statement e viene successivamente passato alla corretta funzione di generazione del codice.

La definizione di una funzione è compilata come un metodo Java privato e statico della classe principale in modo da poter essere chiamato senza il

bisogno di istanziare un oggetto. Il tipo di ritorno è mappato ad un tipo Java mediante il metodo `toJavaType()` mentre ogni parametro è compilato separatamente dall'apposito metodo `compile()` nella forma *tipo* seguito dal *nome*.

Le chiamate a funzione sono invece tradotte in chiamate ai metodi statici precedentemente descritti. All'interno della `compile()` si è tenuto conto della conversione di tipo da `integer` a `real` di ogni parametro.

Una dichiarazione di variabile è sostanzialmente tradotta come una normale dichiarazione nel linguaggio Java. All'interno del metodo si tiene conto della natura dell'assegnamento, ossia se si tratta di tipi primitivi oppure di liste o insiemi. Nel primo caso in particolare viene trattata in modo differente la situazione di variabile con tipo dedotto dall'espressione identificata dalla parola chiave `var`. Infine è stato necessario tenere conto di casi particolari di variabili a cui viene assegnato un valore di tipo differente ma comunque compatibile.

La compilazione del costrutto di assegnamento invece deve poter tener conto, a differenza delle dichiarazioni di variabili, della possibilità di assegnamenti multipli. Per questo motivo la traduzione avviene in modo ricorsivo tenendo separati l'*lvalue* e l'*rvalue*. In un assegnamento multiplo si avrà infatti che la parte destra non sarà un'espressione, ma sarà a sua volta un assegnamento. Nel caso in cui la parte destra sia invece un'espressione, essa verrà compilata dal relativo metodo mediante una sola chiamata a `toString()` che si occupa di trasformare l'espressione in una stringa.

Similmente a quanto avviene per il metodo `checkType()` discusso precedentemente, anche questa `toString()` ha una struttura ricorsiva che rispecchia la gerarchia dell'albero di sintassi astratta. I tipi primitivi semplici `integer`, `real` e `string` sono compilati come oggetti rispettivamente delle classi `BigInteger`, `BigDecimal` [15, p. 46], `String`. I valori booleani vengono semplicemente tradotti nelle parole chiave Java '`true`' e '`false`' mentre la creazione di liste e insiemi vuoti vengono tradotte nella chiamata ai costruttori rispettivamente della classe `ArrayList` e `LinkedHashSet` di Java. Liste e insiemi non vuoti sono gestiti invece dalle funzioni predefinite del linguaggio `__createList__()` (paragrafo 3.5.1) e `__createSet__()`.

Una espressione di contenimento (paragrafo 3.2.8) è stata tradotta come una chiamata al metodo Java `contains()`, mentre un'espressione *size* è stata delegata al metodo `size()` presente in entrambe le classi `ArrayList` e `LinkedHashSet`. Per l'accesso indicizzato ad un elemento di una lista è stato semplicemente scelto il metodo `get()` [16].

Parlando di operazioni binarie è stata utilizzata una funzione per il *dispatch* chiamata `getCorrectOperation()` che, a seconda dell'operatore binario che viene analizzato in un'espressione, chiama il metodo corretto. Ad

esempio se l'operatore tra due espressioni (di qualsiasi tipo) è '+', verrà chiamato il metodo `addition()`, oppure se è '=' verrà chiamato il metodo `equality()` e così via. Al loro interno ci si occupa di verificare tra quale tipo di espressioni sta avvenendo l'operazione. Si è infatti detto che l'operatore '+' può essere utilizzato ad esempio tra interi o tra liste, con significati differenti in entrambi i casi. Nel primo caso verrà utilizzato il metodo `add()` della classe `BigInteger`, mentre nel secondo caso si delega ulteriormente alla funzione predefinita del linguaggio `__addLists__()` che al suo interno sfrutta il metodo `addAll()` della classe `ArrayList`.

I costrutti decisionali *if-else* e *while* non differiscono sostanzialmente da quelli presente in Java se non per l'assenza di parentesi graffe come delimitatori di un blocco. Nel paragrafo 3.1.1 abbiamo infatti discusso la particolare struttura a blocchi del linguaggio la cui traduzione è molto semplice e consiste nel racchiudere tra parentesi graffe la compilazione di ogni statement appartenente al blocco. In entrambi i costrutti comunque l'espressione condizionale viene compilata ed inserita fra parentesi tonde.

La traduzione del costrutto *for-each* rispecchia la sintassi di quello Java presente dalla versione 5 del linguaggio. Per fare ciò è stato necessario all'interno della `compile()` risalire al tipo della lista o dell'insieme da scandire per estrarre il suo *simple type* tramite il metodo `toStringSimpleType()`. Ad esempio il simple type di una lista di interi è `BigInteger` mentre quello di un insieme di stringhe è `String`.

La compilazione del costrutto di unificazione si compone di due parti che vanno a delineare due statement Java. Nella prima parte viene tradotto l'assegnamento della testa della lista *target* alla variabile *head* tramite l'uso del metodo `get(0)` che appunto estrae il primo elemento. Nella seconda invece viene utilizzato il metodo `sublist()` per ritornare una lista privata della sua testa.

Infine, per quanto riguarda la stampa a video, la traduzione in Java dei due costrutti `print()` e `println()` avviene utilizzando rispettivamente i metodi `System.out.print()` e `System.out.println()`. A questi due metodi viene passato come parametro l'espressione compilata.

3.5.1 Funzioni predefinite del linguaggio

Avendo definito le operazioni fra tipi in modo differente dal linguaggio Java e dovendo occuparsi delle conversioni di tipo, è stato d'aiuto per la generazione del codice affidarsi a metodi ausiliari chiamati *funzioni predefinite del linguaggio* riconoscibili per la presenza ad inizio e fine metodo di un doppio *underscore*. Si noti che questi metodi non appartengono al linguaggio in senso stretto e non devono essere utilizzati da chi scrive programmi nel linguaggio

FALL1 poiché il loro scopo è solamente quello di supporto alla traduzione in Java. Prendiamo come esempio il codice:

```
1 list of string abcd -> ["a", "b", "c"]
2 abcd -> abcd + ["d"]
```

L'espressione `[a, b, c]` viene tradotta come

```
1 __createList__(String.class, __toString__( "a"), __toString__(
    "b"), __toString__( "c"))
```

I metodi `__createList__()` (riportata nel listato 3.8) e `__toString__()` sono funzioni predefinite del linguaggio e, mentre la prima si occupa di creare una lista, la seconda prende in input un oggetto di tipo `BigInteger`, `BigDecimal`, `String` o `Boolean` trasformandolo in una stringa. Come si può notare in questo caso si ha codice ridondante dovuto al modo di gestire le conversioni di tipi. Un obiettivo per il raffinamento del linguaggio, oltre a quelli elencati nel paragrafo 4, può essere quello di sistemare tale abuso di notazione.

La seconda espressione è invece un'operazione binaria tra liste di stringhe la cui traduzione è delegata come detto in precedenza al metodo `addition()`. Al suo interno viene utilizzata la funzione predefinita del linguaggio `__addLists__()` che prende come due unici parametri le due liste e ne restituisce una unica concatenando agli elementi della prima quelli della seconda. La traduzione di questa operazione binaria è la seguente:

```
1 __addLists__( abcd,  __createList__(String.class, __toString__(
    "d")))
```

```
1  private static <T> List<T> __createList__(Class<T> classType,
       Object...elements) {
2
3     List<T> resultList = new ArrayList<>();
4
5     for (Object element : elements) {
6
7        Class elementClass = element.getClass();
8
9        if (elementClass == BigInteger.class && classType ==
             BigDecimal.class)
10
11          resultList.add((T) new BigDecimal((BigInteger)element));
12
13       else
14
15          resultList.add((T)element);
16
17    }
18
19    return resultList;
20
21 }
```

Listato 3.8: Il metodo __createList__().

3.6 Outline e labeling

Venendo alle funzionalità che esulano dal linguaggio vero e proprio, in questo lavoro di tesi si è voluto implementare una semplice *outline*, di fatto andando a sostituire il comportamento di default messo a disposizione dall'IDE Eclipse.

Come si può notare infatti in figura 3.4a, di default vengono visualizzate informazioni aggiuntive che per altro sono anche poco significative, perché si basano sulla struttura ad albero del modello. L'idea perciò è stata quella di rendere visibile, così come avviene in Java, solamente l'interfaccia ossia l'insieme dei metodi con le loro *signature*.

Il risultato è dato dalla combinazione delle due classi `FallLabelProvider`

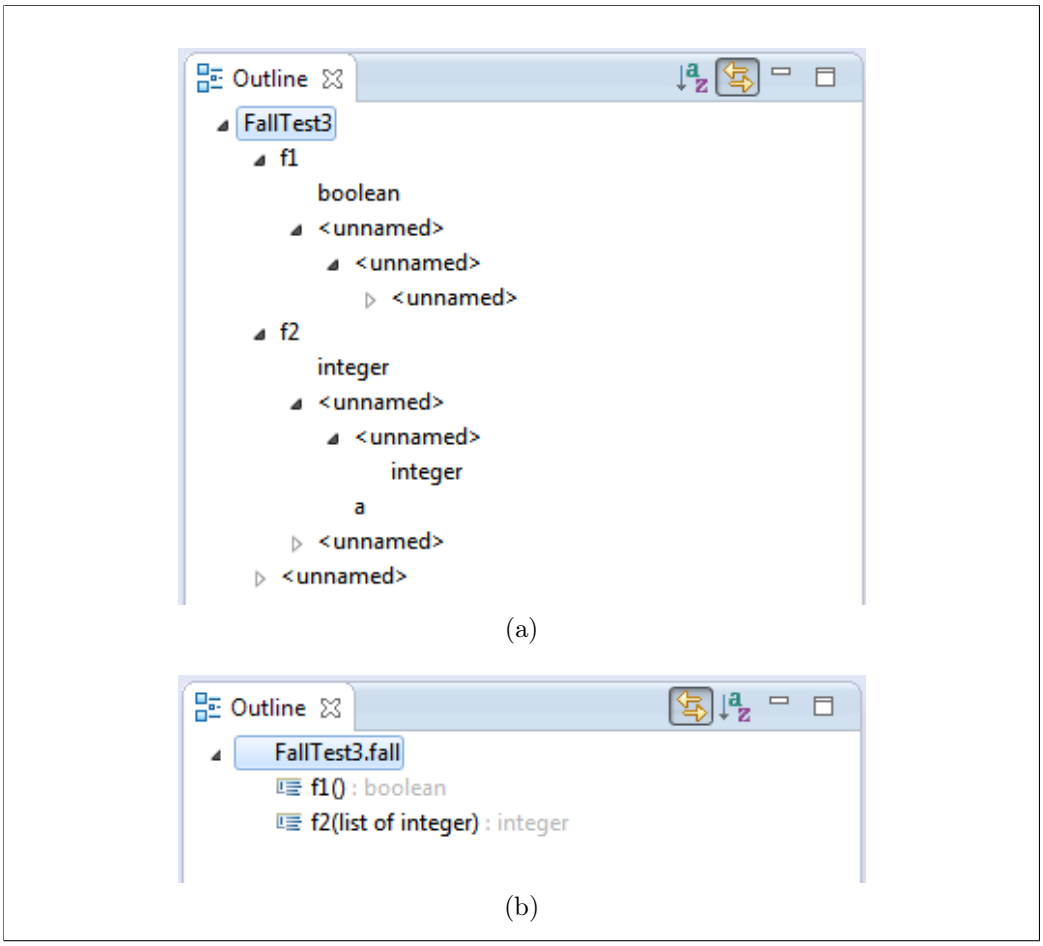

Figura 3.4: La finestra della *outline*.

e `FallOutlineTreeProvider`. Nella prima sono stati utilizzati il metodo `createNode()` per creare il nodo radice, impostando il suo nome al nome del file sorgente e il metodo `_createChildren()` per creare un livello sottostante composto da tutte le funzioni definite.

Nella seconda classe ci si è invece occupati di dare un nome ai nodi delle funzioni. Per fare questo è stato utilizzato il metodo `text()` che prende come parametro un `Object` Java (nel nostro caso un oggetto `FuncDefinition`) e all'interno del quale si è impostato il nome della funzione definita. Oltre ad esso sono stati inclusi i tipi dei parametri formali ed il tipo del valore di ritorno in grigio (si veda la figura 3.4b).

Capitolo 4

Conclusioni

In questo lavoro di tesi si è voluto sperimentare il framework *Xtext* partendo da un'idea approssimativa di linguaggio e cimentandosi man mano con i diversi strumenti messi a disposizione.

Si è partiti dalla definizione delle linee generali e delle caratteristiche che si sarebbero volute nel linguaggio come ad esempio la particolare indentazione, l'uso di tipi di dati a precisione arbitraria (sfruttando le classi fornite da Java) e la necessità di manipolare liste e insiemi. Il procedimento è avvenuto per raffinamenti successivi ma seguendo una logica a cascata composta dalle seguenti fasi:

1. definizione e scrittura della grammatica
2. scoping
3. controllo e verifica dei vincoli
4. generazione del codice

Particolare rilevanza è stata data alla prima fase relativa alla sintassi del linguaggio. Come descritto infatti nel paragrafo 2.2.1, dalla grammatica viene creato il modello ed è proprio quest'ultimo quello su cui si andrà a lavorare in seguito.

La seconda fase è prevista solamente se il linguaggio si discosta a livello di scope dal comportamento di default fornito da *Xtext*, come nel caso del linguaggio FALL1.

Le successive due fasi possono essere svolte in parallelo anche se ci si è presto resi conto che procedere in serie avrebbe reso le cose più facili. Il concetto di fondo è che più controlli verranno eseguiti, più facile sarà generare codice funzionante. Ricordiamo infatti che il linguaggio sviluppato è stato

tradotto in codice Java e che tale codice deve essere esente da errori per far sì che un dato programma possa essere eseguito.

In conclusione *Xtext* si è rivelato uno strumento molto potente anche se relativamente nuovo (la prima versione è stata infatti rilasciata nel 2006). La possibilità di creare con facilità semplici linguaggi, l'integrazione con la *Java Virtual Machine* e la possibilità di implementare generatori di codice automatici ne giustifica infatti l'utilizzo crescente in molte aziende.

Allo stato attuale il linguaggio FALL1 si presenta più che altro come un linguaggio didattico, molto semplice ed effettivamente carente di funzionalità aggiuntive offerte da altri.

Come sviluppo futuro si potrebbe estendere il linguaggio introducendo caratteristiche non presenti come ad esempio la possibilità di importare altri file nel codice sorgente (un comportamento simile agli *import* di Java) gestendo il relativo scope, oppure espandendo i tipi dati.

Ci si può inoltre occupare di aspetti che vanno al di là del linguaggio stesso come per esempio implementare un corretto *content-assist* così come i *quick-fix* relativi ai messaggi di errore.

Un'altra possibilità è quella di seguire l'idea originale di creare un linguaggio molto più ampio che potesse sfruttare gli *agenti*, ad esempio basandosi sul *framework JADE* (*Java Agent DEvelopement*) [17] e permettendo così lo scambio e la gestione di messaggi tramite una rete. Avendo già sviluppato una porzione del linguaggio, quello che può essere implementato in futuro è una specie di *wrapper* che sfrutti il linguaggio FALL1 già implementato e che possa rendere possibile tale compito attraverso l'uso una sintassi chiara e semplice. Questo concetto può anche essere esteso al contesto *mobile*, sfruttando le somiglianze di un'*Activity Android* con un agente JADE.

Appendice A

Esempi di programmi

A.1 Fattoriale

Uno dei classici esempi quando si parla di programmazione è quello che riguarda il fattoriale di un numero. In questo caso `factorial()` è una funzione ricorsiva che prende come parametro un `integer` e restituisce un `integer`.

```
1 integer factorial(integer n)
2   if ( n = 1 )
3     return 1
4   else
5     return n * factorial( n - 1 )
```

A.2 Successione di Fibonacci

Definiti $F_0 = 0$ e $F_1 = 1$, l'n-esimo numero di Fibonacci è dato da $F_n = F_{n-1} + F_{n-2}$. Il seguente esempio si compone di una funzione ricorsiva `fib()` che preso un intero n restituisce l'n-esimo numero di Fibonacci. Il rimanente codice si limita a stampare a video la sequenza composta dai primi dieci numeri della successione ossia:

$$1,\ 1,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55$$

```
integer fib(integer n)
   if (n = 0)
      return 0
   if (n = 1)
      return 1
   return fib(n - 1) + fib(n - 2)


integer n -> 10
integer i -> 1

while (i <= n - 1)
   print("" + fib(i) + ", ")
   i -> i + 1

println(fib(n))
```

Una semplice versione iterativa per calcolare l'n-esimo numero di Fibonacci può essere implementata come segue:

```
integer fib_iter(integer n)
   integer a -> 0
   integer b -> 1
   integer i -> 0

   while (i < n)
      b -> a + b
      a -> b - a
      i -> i + 1

   return a
```

Il seguente codice contiene invece l'implementazione di un'ulteriore versione iterativa che però in questo caso utilizza le liste.

```
integer fib_iter2(integer n)
   if (n < 2)
      return n

   list of integer l -> [0, 1]

   integer i -> 2

   while (i <= n)
      l -> l + [l[i-1] + l[i-2]]
      i -> i + 1

   return l[n]
```

A.3 Bubble sort

Il *bubble sort* è un algoritmo di ordinamento che nel nostro caso opera su liste di interi. Tra tutti gli algoritmi di ordinamento non è però il più efficiente, avendo come complessità computazionale $O(n^2)$. Il codice si compone di due funzioni e di una parte relativa al test dell'algoritmo. La prima funzione illustrata è una funzione ausiliaria chiamata `swap()` che prende in input tre parametri ed il cui codice è riportato nel listato seguente:

```
integer swap(list of integer l, integer i, integer j)
   integer tmp -> l[i]
   l[i] -> l[j]
   l[j] -> tmp
   return 0
```

Il suo scopo è semplicemente quello di scambiare l'elemento i-esimo con l'elemento j-esimo della lista `l` affidandosi all'uso di una variabile temporanea `tmp`. L'implementazione dell'algoritmo vero e proprio, così come la parte relativa al test di funzionamento è riportata invece nel seguente listato:

```
integer bubbleSort(list of integer l)
   boolean swapped -> true
   integer n -> |l| - 1
   while (swapped and n > 0)
      swapped -> false
      integer i -> 0
      while (i < n)
         if (l[i] > l[i+1])
            swap(l, i, i+1)
            swapped -> true
         i -> i+1
      n -> n-1
   return 0

list of integer l -> [10, 9, 5, 2, 1, 6, 8, 3, 4, 7]
bubbleSort(l)
println(l)
```

Ciò che viene visualizzato a video è:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

A.4 Merge sort

Un altro celebre algoritmo di ordinamento che si vuole illustrare è il *merge sort* che si compone di una funzione `mergeSort()` illustrata nel listato seguente:

```
list of integer mergeSort(list of integer l, integer left, integer
    right)
   if (left < right)
      integer center -> left + (right - left) / 2
      mergeSort(l, left, center)
      mergeSort(l, center+1, right)
      merge(l, left, center, right)
   return l
```

e della successiva `merge()` seguita da istruzioni di test:

```
integer merge(list of integer l, integer left, integer center,
    integer right)
   list of integer tmp -> []

   integer index -> 0

   while (index < |l|)
      tmp -> tmp + [0]
      index -> index + 1

   index -> left

   while (index <= right)
      tmp[index] -> l[index]
      index -> index + 1

    integer i -> left
    integer j -> center + 1
    integer k -> left

    while (i <= center and j <= right)
      if (tmp[i] <= tmp[j])
        l[k] -> tmp[i]
        i -> i + 1
      else
        l[k] -> tmp[j]
        j -> j + 1

      k->k+1

    while (i <= center)
      l[k] -> tmp[i]
      k -> k + 1
      i -> i + 1

   return 0

list of integer l -> [10, 9, 5, 2, 1, 6, 8, 3, 4, 7]

println(mergeSort(l,0,|l|-1))
```

La prima si occupa della chiamata ricorsiva e scompone la lista in parti via via più piccole, la seconda invece è dedicata all'ordinamento. Insieme vanno a delineare il celebre approccio *divide et impera*. Il risultato in output sarà una lista di elementi ordinati da uno a dieci.

A.5 Intersezione

L'esempio che si vuole esaminare è l'intersezione tra due liste, ossia l'insieme degli elementi comuni. Il codice è il seguente:

```
1  set of integer intersection(list of integer a, list of integer b)
2     set of integer result -> {}
3        for y in b
4           if (y in a)
5              result -> result + {y}
6     return result
7
8
9  list of integer l -> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
10 list of integer t -> [3, 3, 1, 2, 10, 10, 1]
11
12 println(intersection(l,t))
```

L'algoritmo è sviluppato utilizzando la funzione `intersection()` che prende in input le due liste e restituisce un insieme.

All'interno della funzione viene scandito ogni elemento in `b` tramite un ciclo *for-each* e viene utilizzata un'espressione di contenimento per verificare l'appartenenza di un elemento alla lista `a`. In caso affermativo tale elemento viene aggiunto all'insieme degli elementi appartenenti all'intersezione. Si noti che avendo come valore di ritorno un insieme, l'ordine di apparizione degli elementi è casuale. L'insieme stampato a video sarà:

```
[3, 1, 2, 10]
```

Bibliografia

[1] Agostino Dovier e Roberto Giacobazzi. *Fondamenti dell'Informatica: Linguaggi Formali, Calcolabilità e Complessità.*

[2] Sven Efftinge. *Parsing Expressions with Xtext.* URL: `http://blog.efftinge.de/2010/08/parsing-expressions-with-xtext.html`.

[3] Terence Parr e Kathleen S. Fisher. *LL(*): The Foundation of the ANTLR Parser Generator.* URL: `www.antlr.org/papers/LL-star-PLDI11.pdf`.

[4] *Xtext Tutorial and Documentation.* URL: `http://www.eclipse.org/Xtext/documentation.html`.

[5] *Block scope, validation, linking and content assist.* URL: `http://www.eclipse.org/forums/index.php/m/741529/`.

[6] *20 Facts about Xtend.* URL: `http://jnario.org/org/jnario/jnario/documentation/20FactsAboutXtendSpec.html`.

[7] Sven Efftinge. *Xbase - A new programming language?* URL: `http://blog.efftinge.de/2010/09/xbase-new-programming-language.html`.

[8] *Google Guice Wiki.* URL: `http://code.google.com/p/google-guice/w/list`.

[9] *Bug 401917 - [Outline/XtextReconcilerJob] IAE Error refreshing outline in Editor.* URL: `https://bugs.eclipse.org/bugs/show_bug.cgi?id=401917`.

[10] *Python-like indentation and editor problems.* URL: `http://www.eclipse.org/forums/index.php/m/1053203/`.

[11] Sven Efftinge. *Parsing Expressions with Xtext.* URL: `http://blog.efftinge.de/2010/08/parsing-expressions-with-xtext.html`.

[12] Bruce Eckel. *Thinking in Java. Tecniche avanzate, 4a edizione.*

[13] Maurizio Gabbrielli e Simone Martini. *Linguaggi di programmazione. Principi e paradigmi.* 2006.

[14] James Gosling, Bill Joy, Guy Steele, Gilad Bracha e Alex Buckley. *The Java® Language Specification. Java SE 7 Edition*. 28 Feb. 2013. URL: `http://docs.oracle.com/javase/specs/jls/se7/html/jls-6.html#jls-6.3`.

[15] Bruce Eckel. *Thinking in Java. I fondamenti, 4a edizione.*

[16] *Java™ Platform, Standard Edition 7 API Specification.* URL: `http://docs.oracle.com/javase/7/docs/api/overview-summary.html`.

[17] *Jade - Java Agent DEvelopment Framework.* URL: `http://jade.tilab.com/`.