

Marco Raminella

Esempi di DSL con Xtext e Xtend

Questo è un estratto delle slide che ho prodotto per il corso di Linguaggi e Modelli Computazionali tenuto dal Prof. Enrico Denti presso l'Università di Bologna, corso di Ingegneria Informatica.

Si tratta di esempi di due semplici linguaggi, uno per le espressioni e uno per implementare un linguaggio ad oggetti, affrontando i problemi relativi a ciascun linguaggio utilizzando le caratteristiche di Xtext e Xtend.

Le Extension in Xtend

- I **metodi extension** sono "zucchero sintattico": ci permettono di aggiungere nuovi metodi ai tipi esistenti senza doverli modificare, o dover eseguire una chiamata del metodo di cui abbiamo bisogno.
- **Esempio:** invece di chiamare `m(entity)` possiamo estendere la classe di cui fa parte `entity` col metodo `m` e chiamare `entity.m()`
- Molto comodo per le chiamate in cascata: invece di chiamare `metodo1(metodo2(entity))` potremmo scrivere `entity.metodo1().metodo2()`
- Possiamo anche importare metodi static dalle classi Java esistenti:
 - **`import static extension java.util.Collections.*`**
- In Xtext si possono omettere le parentesi `()` nelle chiamate senza parametri: altro zucchero sintattico 😊

Le Extension in Xtend

- I metodi definiti in una classe Xtext possono essere **automaticamente** usati come metodi extension in quella classe:

```
class Esempio {  
    def mioMetodo(List<?> lista){  
        // implementazione  
    }  
  
    def altroMetodo(){  
        val lista = new ArrayList<String>  
        lista.mioMetodo  
    }  
}
```

Le Extension in Xtend

- Questi metodi potranno essere usati per estendere un'altra classe usando la keyword **extension**:

```
class C {  
    extension Eempio e = new Eempio  
  
    def metodo(){  
        val lista = new ArrayList<String>  
        lista.mioMetodo // equivale a e.mioMetodo(lista)  
        lista.altroMetodo // equivale a e.altroMetodo(lista)  
    }  
}
```

- Possiamo anche dichiarare la extension direttamente nel metodo

```
def metodo(){  
    extension Eempio e = new Eempio  
    ...  
}
```

- O anche come parametro:

- **def** metodo(extension Eempio e)

Dependency injection

- La dependency injection è un **design pattern** nato come evoluzione del pattern Factory, dove andiamo a domandare a un oggetto Factory di istanziare di una dipendenza, quindi abbiamo un oggetto che costruisce un altro oggetto.
- Con la Dependency injection invece passiamo le dipendenze dall'esterno, e deleghiamo la creazione dell'oggetto all'esterno, ad esempio a:
 - Un altro oggetto più in alto nella gerarchia
 - Un **dependency injector**: un framework nato appositamente a questo scopo.
- Uno dei maggiori vantaggi della dependency injection è la possibilità di rendere il testing molto più semplice.

Dependency injection e testing in Xtext

- Xtext usa un **dependency injector**, in particolare usa il framework Google Guice per mettere a disposizione per il testing gli oggetti del nostro linguaggio istanziati dal **parser**.
- Xtext predispone delle classi di testing, in cui gli strumenti che ci mette a disposizione (dependency injection ed extensions) ci renderanno il testing molto più pratico
- È possibile testare le regole, il comportamento e la struttura generata dal nostro linguaggio ancora prima di metterlo all'opera.
- Oltre al linguaggio stesso, potremo testare qualunque cosa: il validatore, il generatore di codice, la UI..
- Tutte le classi di test sono presenti nel sotto-progetto .tests

..Testing del linguaggio..

Per ora testiamo semplicemente se il nostro linguaggio accetta delle espressioni..

Xtext inietta di default il ParseHelper che ha costruito con il nostro DSL

Estendiamo la nostra classe con le Assertion già esistenti in Java

```
Esempio1.xtext  Esempio1ParsingTest.xtend
8  import org
9  import org
10
11  import org
12  import org.junit
13  import org.xtext.example.mydsl.esempio1.ExprModel
14  import static extension org.junit.Assert.*
15  @RunWith(XtextRunner)
16  @InjectWith(Esempio1InjectorProvider)
17  class Esempio1ParsingTest {
18      @Inject extension
19      ParseHelper<ExprModel> parseHelper
20
21  @Test
22  def void testExpr(){
23      "3 + 2".parse.assertNotNull
24  }
25
```

Finished after 1,35 seconds

Runs: 3/3 Errors: 0 Failures: 0

org.xtext.example.mydsl.tests.Esempio1

- testAll (0,152 s)
- testExpr (0,003 s)
- testTerm (0,003 s)

Personalizzazione dell'IDE: label

- Xtext predispone una serie di strumenti che ci saranno utili per personalizzare l'IDE che verrà costruito sopra il nostro linguaggio.
- Un primo strumento molto semplice sono le **labels**: queste serviranno sia nella Outline (che mostrerà l'AST degli oggetti che andremo a definire con il nostro DSL) sia nel content-assist.
- Nel sub-package **ui.labeling** troveremo tutto quello che serve a gestire le label

..aggiungiamo le Label..

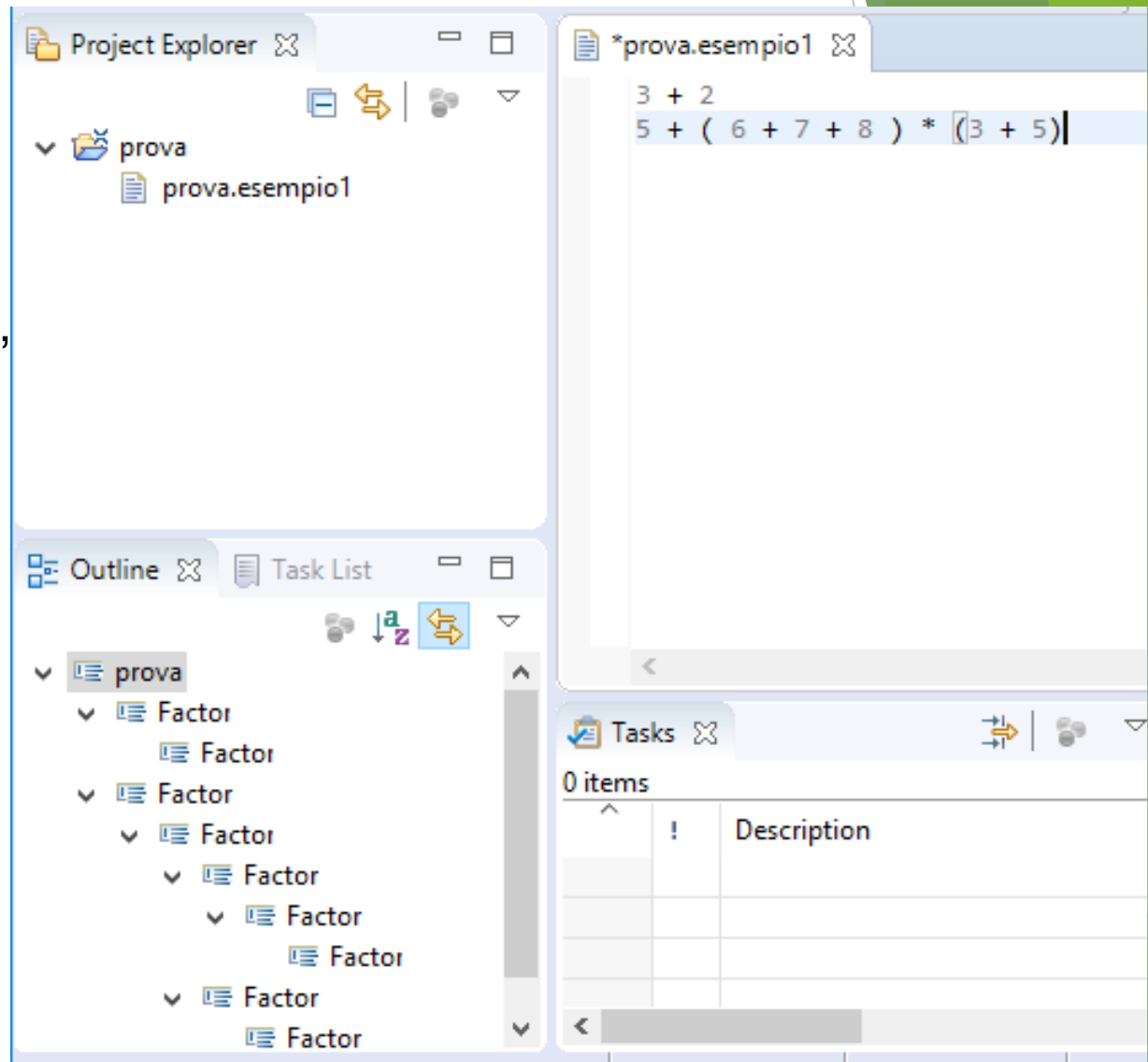
```
20 * generated by Xtext 2.13.0
4 package org.xtext.example.mydsl.ui.labeling
5
6 import com.google.inject.Inject
12
13 /**
14  * Provides labels for EObjects.
15  *
16  * See https://www.eclipse.org/Xtext/documentation/304\_ide\_concepts.html#label-provider
17  */
18 class Esempio1LabelProvider extends DefaultEObjectLabelProvider {
19
20     @Inject
21     new(AdapterFactoryLabelProvider delegate) {
22         super(delegate);
23     }
24
25     def text(Expr exp){
26         "Expr"
27     }
28
29     def text (Term term){
30         "Term"
31     }
32
33     def text (Factor factor){
34         "Factor"
35     }
36 }
```

Estensione del provider
predefinito di label!

Per ogni oggetto del nostro DSL,
definiamo con quale testo
«etichettarlo»

..aggiungiamo le Label..

A causa del loop (il warning ci aveva avvisato!) tutti gli elementi vengono sovrascritti da Factor, quindi l'outline non avrà molto senso!



Terzo esempio: espressioni con AST

```
1 grammar it.unibo.disi.enricodenti.Esempio2 with org.eclipse.xtext.
2
3 generate esempio2 "http://www.unibo.it/disi/enricodenti/Esempio2"
4
5 Model:
6     prog+=Expr*
7 ;
8
9 Expr:
10     Term (
11         ({PlusExp.left = current} '+'
12         | {MinusExp.left = current} '-')
13         t2=Term)*
14 ;
15
16 Term :
17     Factor (
18         ({TimesExp.left = current} '*'
19         | {DivExp.left = current} ':' )
20         f2=Factor)*
21 ;
22
23 Factor:
24     valore = INT |
25     '(' expr = Expr ')'
26 ;
27
```

Con la notazione *{nome tipo}* possiamo esplicitare la creazione di un elemento nell'AST

Terzo esempio: espressioni con AST

```
generate esempio2 "http://www.unibo.it/disi/enricc
```

```
Model:
```

```
  prog+=Expr*
```

```
;
```

```
Expr returns Exp:
```

```
  Term (|
```

```
    ({PlusExp.left = current} '+'
```

```
    | {MinusExp.left = current} '-' )
```

```
  t2=Term)*
```

```
;
```

```
Term returns Exp:
```

```
  Factor (
```

```
    ({TimesExp.left = current} '*'
```

```
    | {DivExp.left = current} ':' )
```

```
  f2=Factor)*
```

```
;
```

```
Factor returns Exp:
```

```
  valore = INT |
```

```
  '(' expr = Expr ')'
```

```
;
```

Anche con *returns* possiamo esplicitare la creazione di un oggetto nell'AST

Accumula il risultato a sinistra

Testing del DSL

- Per testare questo DSL andremo prima a creare due funzioni aggiuntive che stamperanno la rappresentazione verbosa del nostro linguaggio, per poi confrontarla e vedere se corrisponde a quello che ci aspettiamo:

```
sempio2ParsingTest.xtend  Esempio2LabelProvider.xtend  Esempio2.xtext

import it.unibo.disi.enricodenti.esempio2.DivExp
import it.unibo.disi.enricodenti.esempio2.Value
import static extension org.junit.Assert.*
@RunWith(XtextRunner)
@InjectWith(Esempio2InjectorProvider)
class Esempio2ParsingTest {
    @Inject extension
    ParseHelper<Model> parseHelper

    def private String stringRepr(Exp e) {
        switch(e){
            PlusExp:
                '''(«e.left.stringRepr» + «e.t2.stringRepr）」'''
            Value:
                '''«e.valore»'''
        }.toString
    }
}
```

Metodo per costruire la rappresentazione verbosa degli elementi

Testing del DSL

- Per testare questo DSL andremo prima a creare due funzioni aggiuntive che stamperanno la rappresentazione verbosa del nostro linguaggio, per poi confrontarla e vedere se corrisponde a quello che ci aspettiamo:

```
def private assertRepr(CharSequence input, CharSequence expected){  
    (input).parse => [expected.assertEquals(  
        prog.last.stringRepr  
    )  
    ]  
}  
  
@Test  
def void testPlus(){  
    "10 + 5 + 1 + 2".assertRepr("(((10 + 5) + 1) + 2)");  
}  
}
```

Metodo per
comparare la
rappresentazione
e creata con
stringRepr con
quella creata
dal parser..

Proviamo a vedere se la rappresentazione
nell'AST corrisponde a cosa ci aspettavamo
(ricorsione sinistra)..

Labels

Dato che abbiamo definito esplicitamente la creazione degli oggetti, proviamo a vedere come funzionano le label. Sul sotto progetto ui, nella classe Xtend Esempio3LabelProvider, andiamo a definire le label per gli oggetti appena creati:

```
19 class Esempio2LabelProvider extends DefaultEObjectLabelProvider {
20
21     @Inject
22     new(AdapterFactoryLabelProvider delegate) {
23         super(delegate);
24     }
25
26     def text(PlusExp exp){
27         "+"
28     }
29
30     def text(MinusExp exp){
31         "-"
32     }
33
34     def text(TimesExp exp){
35         "*"
36     }
37
38     def text(DivExp exp){
39         ":"
40     }
41 }
```

..Labels

Project Ex...

esempio2

prova

*prova.esempio2

```
1 + 2 + 4 + 3 + 5
6 + 3 : ( 2 - 4 )
```

AST risultante

prova

+ + + + +
















: <unnamed>

Per cambiare anche questa <unnamed> è sufficiente assegnare un nome all'oggetto (Expr), come abbiamo fatto col resto!

Eclipse Modeling Framework (EMF)

Per rappresentare l'AST dei linguaggi che vengono definiti Xtext utilizza un framework di Eclipse creato a questo scopo. In fase di generazione degli Xtext artifacts viene costruita automaticamente una serie di classi Java che fanno parte del modello EMF.

Utilizzeremo delle funzioni utility dell'EMF in fase di validazione per estrapolare informazioni sull'AST degli oggetti che vengono istanziati con la grammatica che abbiamo definito.

- ▼  src-gen
 - >  org.example.expressions
 - ▼  org.example.expressions.expressions
 - >  AbstractElement.java
 - >  And.java
 - >  BoolConstant.java
 - >  Comparison.java
 - >  Equality.java
 - >  EvalExpression.java
 - >  Expression.java
 - >  ExpressionsFactory.java
 - >  ExpressionsModel.java
 - >  ExpressionsPackage.java
 - >  IntConstant.java
 - >  Minus.java

Classi Java di EMF
generate da Xtext

Expressions DSL

Tratto da Implementing Domain-Specific Languages with Xtext and Xtend,
second edition, Lorenzo Bettini, packtpub

La versione completa e ottimizzata si trova su GitHub (il link come riferimento è in calce), nel corso di questa esercitazione costruiremo passo passo una versione semplificata (anche rispetto al libro) di questo esempio.

<https://github.com/LorenzoBettini/packtpub-xtext-book-2nd-examples>

Esempio completo: Expressions DSL

Grammatica di base EBNF:

```
<AbstractElement>::=
```

```
<Variable> | <EvalExpression>;
```

```
<Variable>::=
```

```
'var' {<ID>} '=' <Expression>;
```

```
<EvalExpression>::=
```

```
'eval' <Expression>;
```

```
<Expression>::=
```

```
{<IntConstant>} | {<StringConstant>} | {<BoolConstant>} |  
{<VariableRef>;}
```

Esempio completo: Expressions DSL

Definizione con Xtext (usa i tipi di `org.eclipse.xtext.common.Terminals`) :

Expression returns Atomic:

```
‘(‘ Expression ‘)’ | {IntConstant} value=INT | {StringConstant} value=STRING  
| {BoolConstant} value=(‘true’ | ‘false’) | {VariableRef}  
variable=[Variable];
```

Con le parentesi quadre facciamo riferimento a una istanza esistente di un oggetto Variable

Aggiungiamo la somma:

Expression:

```
Atomic({Plus.left=current} ‘+’ right=Atomic)*;
```

Esempio completo: Expressions DSL

<https://github.com/LorenzoBettini/packtpub-xtext-book-2nd-examples>

```
grammar org.example.expressions.Expressions with org.eclipse.xtext.common.Terminals
```

```
generate expressions "http://www.example.org/expressions/Expressions"
```

```
ExpressionsModel:
```

```
    elements+=AbstractElement*;
```

```
AbstractElement:
```

```
    Variable | EvalExpression;
```

```
Variable:
```

```
    'var' name=ID '=' expression=Expression;
```

```
EvalExpression:
```

```
    'eval' expression=Expression;
```

```
Expression:
```

```
    Atomic({Plus.left = current} '+' right=Atomic)* ;
```

```
Atomic returns Expression:
```

```
    {IntConstant} value=INT |
```

```
    {StringConstant} value=STRING |
```

```
    {BoolConstant} value=('true' | 'false') |
```

```
    {VariableRef} variable=[Variable];
```

Dei primi test..

Abbiamo una prima base su cui possiamo già fare dei test per provare la grammatica che abbiamo appena scritto.

Facciamo costruire gli Xtext artifacts, dopodichè passiamo al progetto .tests. Vogliamo testare:

- ▶ Se le espressioni vengono **riconosciute** come ci aspettiamo
- ▶ Se le variabili *var* che abbiamo introdotto vengono riconosciute correttamente dal nostro stesso linguaggio
- ▶ Se la somma è ricorsiva a sinistra, ovvero se l'AST è costruito correttamente.

..Dei primi test..

```
def void testEvalExpression(){
    "eval 10".parse.assertNotNull
}
@Test
def void testEvalBoolean(){
    "eval false".parse.assertNotNull
}
@Test
def void testVariable(){
    "var i = 10".parse.assertNotNull
}
@Test
def void testBoolean(){
    "var i = true".parse.assertNotNull
}
@Test
def void testString(){
    "var i = \"mario\"".parse.assertNotNull
}

@Test
def void testVariableReference(){
    ...
    var i = 10
    eval i
    ''' .parse => [(elements.last.expression as VariableRef).variable.assertSame(elements.head)]
}
```

Ricordiamoci di aggiungere le Assertion come extension e di iniettare il ParseHelper, come avevamo visto prima!

Proviamo a valutare le varie espressioni che abbiamo definito col nostro linguaggio, e a vedere se vengono riconosciute..

Qui proviamo a vedere se il riferimento alla variabile è corretto: l'ultima espressione (eval i) corrisponde al primo elemento?

Lambda expression!

..Dei primi test

```
@Test
def void testParenthesis() {
  "eval (10)".parse.elements.get(0) as IntConstant
}
```

```
@Test
def void testPlus() {
  "10 + 5 + 1 + 2".assertRepr("(((10 + 5) + 1) + 2)")
}
```

```
@Test def void testPlusWithParenthesis() {
  "( 10 + 5 ) + ( 1 + 2 )".assertRepr("((10 + 5) + (1 + 2))")
}
```

```
def String stringRepr(Expression e) {
  switch (e) {
    Plus:
      '''(«e.left.stringRepr» + «e.right.stringRepr）」'''
    IntConstant: '''«e.value»'''
    StringConstant: '''«e.value»'''
    BoolConstant: '''«e.value»'''
    VariableRef: '''«e.variable.name»'''
  }
}
```

```
def assertRepr(CharSequence input, CharSequence expected) {
  ("eval " + input).parse => [
    assertNoErrors;
    expected.assertEquals(
      elements.last.expression.stringRepr
    )
  ]
}
```

Proviamo sia le parentesi..

... che la rappresentazione nell'AST
sia con la corretta gerarchia (a
destra)

Questa è la stessa utility di prima, fatta
cambiando il nome degli oggetti..

Questa è leggermente diversa: esplicitiamo il fatto
che andiamo a prendere solo la parte Expression,
rimuovendo il resto!

Aggiungiamo altri operatori

Vogliamo aggiungere alla nostra grammatica anche sottrazioni, moltiplicazioni e divisioni:

```
Expression:
    PlusOrMinus
;

PlusOrMinus returns Expression:
    Atomic (
        ({Plus.left = current} '+' | {Minus.left=current} '-')
        right = MulOrDiv
    )*
;

MulOrDiv returns Expression:
    Atomic (
        ({MulOrDiv.left = current} op=('*' | '/'))
        right=Atomic
    )*
;
```

Uguale al secondo esempio, viene esplicitata la creazione degli oggetti: nell'AST ci saranno Expression, Plus e Minus (non PlusOrMinus)

Per le moltiplicazioni e le divisioni adottiamo un approccio diverso:

- Sarà creato un oggetto MulOrDiv per ogni moltiplicazione o divisione.
- Starà a noi, in fase di parsing, andare a vedere cosa c'è nella feature *op*.
- Si tratta di un approccio diverso, andremo a vedere vantaggi e svantaggi nella fase di valutazione.

Testing dei nuovi operatori

```
def String stringRepr(Expression e) {  
  switch (e) {  
    Plus:  
      '''(«e.left.stringRepr» + «e.right.stringRepr»)'''  
    Minus:  
      '''(«e.left.stringRepr» - «e.right.stringRepr»)'''  
    MulOrDiv:  
      '''(«e.left.stringRepr» «e.op» «e.right.stringRepr»)'''  
    IntConstant: '''«e.value»'''  
    String:  
    BoolCo  
    VariableRef: «e.variable.name»  
  }  
}
```

Usiamo la feature op

Aggiorniamo la stringRepr per rappresentare i nuovi oggetti..

...e testiamo se la precedenza fra gli operatori è corretta:

```
@Test  
def void testPlusMulPrecedence(){  
  "10 + 5 * 2 - 5 / 1".assertRepr("((10 + (5 * 2)) - (5 / 1))")  
}
```

Aggiungiamo le espressioni booleane e algebriche

Per concludere, aggiungiamo le espressioni booleane e algebriche. Dobbiamo tenere conto della precedenza con cui devono essere valutate:

1. **or** booleano: operatore `||`
2. **and** booleano: operatore `&&`
3. **eguaglianza** e **diseguaglianza**: operatori `==` e `!=`
4. **comparazioni**: `<`, `<=`, `>`, `>=`
5. addizioni e sottrazioni
6. moltiplicazioni e divisioni

Aggiungiamo le espressioni booleane e algebriche

Aggiungiamo quindi gli elementi alla nostra grammatica, prima di Expression, secondo l'ordine definito:

```
Expression: Or;  
Or returns Expression:  
    And (  
        {Or.left=current} "||" right=And  
    );  
And returns Expression:  
    Equality (  
        {And.left=current} "&&" right=Equality  
    );  
Equality returns Expression:  
    Comparison (  
        {Equality.left=current} op=("==" | "!=")  
        right=Comparison  
    );  
Comparison returns Expression:  
    PlusOrMinus (  
        {Comparison.left=current} op(">=" | "<=" | ">"  
        right=PlusOrMinus  
    );
```

<https://github.com/LorenzoBettini/packtpub-xtext-book-2nd->

1. or booleano: operatore ||
2. and booleano: operatore &&
3. eguaglianza e disuguaglianza: operatori == e !=
4. comparazioni: <, <=, >, >=
5. addizioni e sottrazioni
6. moltiplicazioni e divisioni

Aggiungiamo le espressioni booleane e algebriche

Modifichiamo leggermente anche l'oggetto foglia, per includere la negazione !:

```
Atomic returns Expression:  
  '(' Expression ')' |  
  {Not} "!" expression=Atomic |  
  {IntConstant} value=INT |  
  {StringConstant} value=STRING |  
  {BoolConstant} value=('true'|'false') |  
  {VariableRef} variable=[Variable]  
;
```

<https://github.com/LorenzoBettini/packtpub-xtext-book-2nd-examples>

Testing: come prima, **come esercizio** aggiorniamo la StringRepr e testiamo delle espressioni complesse che usano la grammatica aggiornata!

- Se stiamo aggiornando una grammatica, è consigliabile provare i nuovi elementi prima da soli e poi insieme al resto della grammatica

Type checking

Un primo passo verso la valutazione è il type checking. Il tipo delle nostre espressioni dipende dalla **semantica** che vogliamo dare agli elementi del nostro DSL.

In Xtext siamo noi a determinare quale sarà la politica di gestione dei tipi del nostro DSL, sulla base di come lo andremo a implementare.

Per l'esempio che stiamo trattando, vogliamo realizzare un type system che abbia questo comportamento:

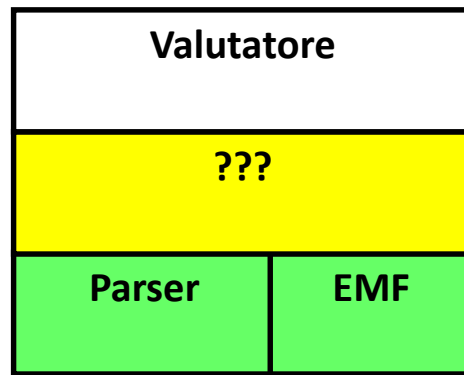
- ▶ In una espressione Plus, se uno dei due operatori è di tipo stringa, allora l'intera espressione è di tipo stringa (concatenazione). Se invece entrambi gli operatori sono di tipo INT, allora l'intera espressione sarà un intero (somma aritmetica).
- ▶ Non sono accettate Plus fra valori booleani (sarebbe ambiguo).
- ▶ Equality può essere valutata solo per sotto-espressioni dello stesso tipo
- ▶ Comparison può operare solo in sotto-espressioni dello stesso tipo, tranne le espressioni booleane.

Loose type computation, strict type checking

Tipicamente la **validazione** e la **type computation** (la determinazione **del tipo di una frase**) sono implementati insieme. Tuttavia questo comportamento potrebbe fornire informazioni sbagliate agli utenti del nostro DSL. Ad esempio:

$$(1 + 10) < (2 * (3 + "a"))$$

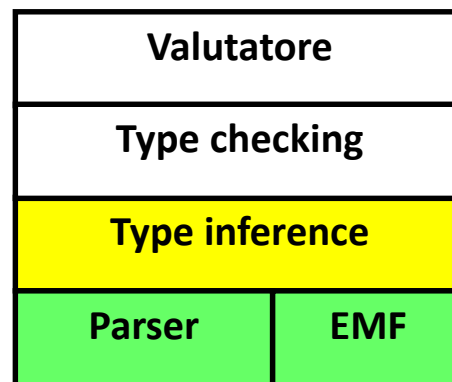
È una espressione non valida, ma solo nella parte $2 * (3 + "a")$:
marchiare l'intera espressione come sbagliata sarebbe contraddittorio.



Loose type computation, strict type checking

Separiamo l'**inferenza del tipo** dalla validazione:

- Realizziamo una classe **ExpressionsTypeComputer** che, per ogni tipo di espressione, andrà ad eseguire la **propria** politica di valutazione del tipo.
- Implementiamo poi le clausole *@Check* nel **validatore** per ogni tipo di espressione (tranne le costanti, che sono già valide), queste clausole andranno a controllare ogni pezzo di espressione con il nostro **ExpressionsTypeComputer**.

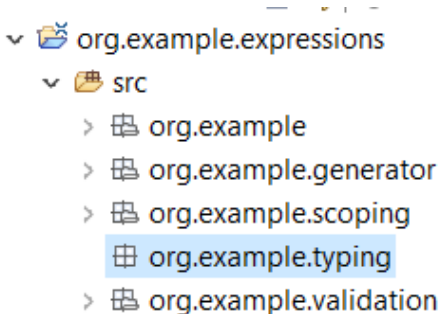


Type Computer

Type computer

Ci serve un **modo** per rappresentare i **tipi** del nostro linguaggio, per fare ciò realizziamo una **tassonomia** di oggetti che li rappresentano.

Creiamo un nuovo package `org.example.expressions.typing` nel progetto base, l'interfaccia `ExpressionsType.xtext` e i tre tipi `IntType`, `StringType`, `BoolType`:



```
1 package org.example.typing
2
3 interface ExpressionsType {
4     override String toString()
5 }
```

Interfaccia ExpressionsType

```
class IntType implements ExpressionsType {
    override toString() { "int" }
}

class StringType implements ExpressionsType {
    override toString() { "string" }
}

class BoolType implements ExpressionsType {
    override toString() { "boolean" }
}
```

Classe IntType

Classe StringType

Classe BoolType

Type computer

La `ExpressionsTypeComputer` definisce un campo *static* per ogni tipo, usando il pattern Singleton si va a confrontare, con l'operatore `===` di Xtext, se una certa istanza di `ExpressionsType` corrisponde all'istanza di quel determinato tipo:

```
class ExpressionsTypeComputer {  
    public static val STRING_TYPE = new StringType  
    public static val INT_TYPE = new IntType  
    public static val BOOL_TYPE = new BoolType  
  
    def isStringType(ExpressionsType type) {  
        type === STRING_TYPE  
    }  
  
    def isIntType(ExpressionsType type) {  
        type === INT_TYPE  
    }  
  
    def isBoolType(ExpressionsType type) {  
        type === BOOL_TYPE  
    }  
}
```

Istanze static

Confronta l'istanza di ExpressionsType

<https://github.com/LorenzoBettini/packtpub-xtext-book-2nd-examples>

Type computer - Polimorfismo in Xtext

A questo punto dobbiamo implementare la logica con cui viene inferito il tipo di una Expression, dalla funzione **typeFor(Expression e)**. Per fare ciò sfruttiamo il pattern **dispatch**. Un metodo dispatch in Xtext permette di realizzare il polimorfismo, andando a cambiare il comportamento di un metodo sulla base dell'implementazione specifica degli oggetti che gli vengono passati.

Cominciamo con le Expression che si possono computare direttamente:

```
def dispatch ExpressionsType typeFor(Expression e) {  
  switch (e) {  
    StringConstant: STRING_TYPE  
    IntConstant: INT_TYPE  
    BoolConstant: BOOL_TYPE  
    Not: BOOL_TYPE  
    MulOrDiv: INT_TYPE  
    Minus: INT_TYPE  
    Comparison: BOOL_TYPE  
    Equality: BOOL_TYPE  
    And: BOOL_TYPE  
    Or: BOOL_TYPE  
  }  
}
```

Type computer - Polimorfismo in Xtext

Con un altro metodo **dispatch** andiamo a inferire di che tipo potrebbe essere una expression Plus, ricordiamo il comportamento che avevamo definito:

In una espressione Plus, se uno dei due operatori è di tipo stringa, allora l'intera espressione è di tipo stringa. Se invece entrambi gli operatori sono di tipo INT, allora l'intera espressione sarà un intero.

Non sono accettate Plus fra valori booleani.

```
def dispatch ExpressionType typeFor(Plus e) {  
  val leftType = e.left.typeFor  
  val rightType = e.right?.typeFor  
  if (leftType.isStringType || rightType.isStringType)  
    STRING_TYPE  
  else  
    INT_TYPE  
}
```

E se stavamo sommando booleani per sbaglio?
È compito del type checker verificare!

Testing del Type computer

Ovviamente vogliamo testare anche il funzionamento del nostro nuovo Type computer. Per fare ciò, innanzitutto, dobbiamo modificare il file MANIFEST.mf del progetto principale, per far sì che vengano generati gli Xtext artifacts anche per le nuove classi che abbiamo creato:

The screenshot shows the Package Explorer on the left and the MANIFEST.MF file on the right. The Package Explorer shows the project structure, with the MANIFEST.MF file highlighted under the META-INF folder. The MANIFEST.MF file content is displayed on the right, showing various bundle properties and package declarations. The 'Import-Package' line is highlighted, and the 'org.example.expressions.validation' package is pointed out by a yellow callout.

Package Explorer:

- org.example.expressions
 - src
 - src-gen
 - xtend-gen
 - JRE System Library [JavaSE-1.8]
 - Plug-in Dependencies
 - META-INF
 - MANIFEST.MF**
 - model
 - build.properties
 - plugin.xml
- org.example.expressions.ide
- org.example.expressions.tests
- org.example.expressions.ui
- org.example.expressions.ui.tests

MANIFEST.MF Content:

```
7 Bundle-ActivationPolicy: lazy
8 Require-Bundle: org.eclipse.xtext,
9   org.eclipse.xtext.xbase,
10  org.eclipse.equinox.common;bundle-version="3.5.0",
11  org.eclipse.emf.ecore,
12  org.eclipse.xtext.xbase.lib;bundle-version="2.13.0",
13  org.antlr.runtime,
14  org.eclipse.xtext.util,
15  org.eclipse.emf.common,
16  org.eclipse.xtend.lib;bundle-version="2.13.0"
17 Bundle-RequiredExecutionEnvironment: JavaSE-1.8
18 Export-Package: org.example.expressions.generator,
19   org.example.expressions.parser.antlr.internal,
20   org.example.expressions.expressions.util,
21   org.example.expressions.services,
22   org.example.expressions.expressions,
23   org.example.expressions.expressions.impl,
24   org.example.expressions,
25   org.example.expressions.scoping,
26   org.example.expressions.serializer,
27   org.example.expressions.parser.antlr,
28   org.example.expressions.validation,
29   org.example.expressions.typing
30 Import-Package: org.apache.log4j
```

Callouts:

- 1: Points to the MANIFEST.MF file in the Package Explorer.
- 2: Points to the 'Import-Package' line in the MANIFEST.MF file.
- 3: Points to the 'org.example.expressions.validation' package in the 'Import-Package' line.

Testing del Type computer

A questo punto (dopo aver rigenerato gli Xtext artifacts) andiamo nel sottoprogetto .tests e creiamo una nuova classe di test TypeComputerTest

```
package org.example.expressions.tests

import com.google.inject.Inject
import com.google.inject.testing.fieldbehavior.injected
import org.junit.Test
import org.junit.runner.RunWith
import org.mockito.Mockito
import org.mockito.junit.MockitoJUnitRunner
import org.xtext.example.expressions.ExpressionsModel
import org.xtext.example.expressions.ExpressionsTypeComputer

@RunWith(MockitoJUnitRunner::class)
@Inject
class ExpressionsTypeComputerTest {

    @Inject extension ParseHelper<ExpressionsModel>
    @Inject extension ExpressionsTypeComputer

    def assertEvalType(CharSequence input, ExpressionsType expectedType) {
        ("eval " + input).assertType(expectedType)
    }

    def assertType(CharSequence input, ExpressionsType expectedType) {
        input.parse.elements.last
            .expression.assertType(expectedType)
    }

    def assertType(Expression e, ExpressionsType expectedType) {
        expectedType.assertSame(e.typeFor)
    }
}
```

Queste due (tre) funzioni fanno buona parte del lavoro:

1. `assertEvalType` chiama la `eval` del nostro linguaggio
2. `assertType` esegue il parsing, dopodichè richiama la propria seconda forma, che va a verificare effettivamente il tipo

Testing del Type computer

Aggiungiamo dei test (questi sono solo alcuni, proviamone altri!):

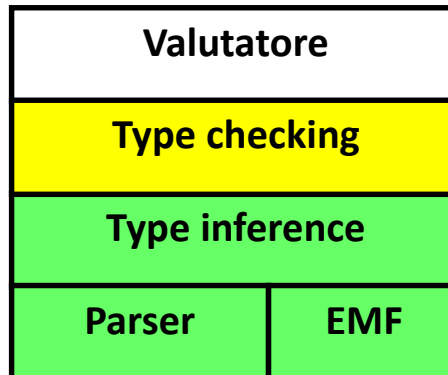
```
@Test def void intConstant() { "10".assertEvalType(INT_TYPE) }
@Test def void stringConstant() { "'foo'".assertEvalType(STRING_TYPE) }
@Test def void boolConstant() { "true".assertEvalType(BOOL_TYPE) }

@Test def void numericPlus(){ "1 + 2".assertEvalType(INT_TYPE) }
@Test def void stringPlus(){ "'a' + 'b'".assertEvalType(STRING_TYPE) }
@Test def void numAndStringPlus(){ "'a' + 2".assertEvalType(STRING_TYPE) }
@Test def void stringAndBoolPlus(){ "1 + true".assertEvalType(INT_TYPE) }
```

Il validatore

Siamo finalmente pronti a mettere mano all'`ExpressionsValidator`, che è stato generato nella sottoclasse `.validation`. Iniettiamo il nostro `ExpressionsTypeComputer` e scriviamo alcuni **metodi utility** per verificare la coerenza dei tipi.

Le **clausole** di controllo vere e proprie le scriveremo poi, in forma compatta, con il tag `@Check`.



Il validatore: metodi utility

```
class ExpressionsValidator extends AbstractExpressionsValidator {  
    protected static val ISSUE_CODE_PREFIX = "org.example.expressions."  
    public static val TYPE_MISMATCH = ISSUE_CODE_PREFIX + "TypeMismatch"  
  
    @Inject extension ExpressionsTypeComputer  
    def private checkExpectedBoolean(Expression exp, EReference reference) {  
        checkExpectedType(exp, ExpressionsTypeComputer.BOOL_TYPE, reference)  
    }  
    def private checkExpectedType(Expression exp, ExpressionsType expectedType, EReference reference) {  
        val actualType = getTypeAndCheckNotNull(exp, reference)  
        if (actualType != expectedType)  
            error("mi aspettavo un " + expectedType + " , ma invece era " + actualType, reference, TYPE_MISMATCH)  
    }  
    def private ExpressionsType getTypeAndCheckNotNull(Expression exp, EReference reference) {  
        val type = exp?.typeFor  
        if (type == null)  
            error("tipo null", reference, TYPE_MISMATCH)  
        return type;  
    }  
    def private checkNotBoolean(ExpressionsType type, EReference reference) {  
        if (type.isBoolType) {  
            error("boolean non va bene qui", reference, TYPE_MISMATCH)  
        }  
    }  
}
```

L'operatore ?. serve ad accettare null e non lanciare eccezioni

Il validatore per le Plus

Per il checking della somma, seguiamo la politica che avevamo definito fin da prima: ○○○

In una espressione Plus, se uno dei due operatori è di tipo stringa, allora l'intera espressione è di tipo stringa. Se invece entrambi gli operatori sono di tipo INT, allora l'intera espressione sarà un intero.

Non sono accettate Plus fra valori booleani.

```
@Check def checkType(Plus plus) {  
  val leftType = getTypeAndCheckNotNull(plus.left, ExpressionsPackage.Literals.PLUS_LEFT)  
  val rightType = getTypeAndCheckNotNull(plus.right, ExpressionsPackage.Literals.PLUS_RIGHT)  
  if (leftType.isIntType || rightType.isIntType ||  
      (!leftType.isStringType && !rightType.isStringType)) {  
    checkNotBoolean(leftType, ExpressionsPackage.Literals.PLUS_LEFT)  
    checkNotBoolean(rightType, ExpressionsPackage.Literals.PLUS_RIGHT)  
  }  
}
```

Operatori destro e sinistro nell'EMF

<https://github.com/LorenzoBettini/packtpub-xtext-book-2nd-examples>

Il validatore

Per not, and ed or controlliamo che le sotto-espressioni siano booleane, passando le eReference di EMF che corrispondono alle parti dell'oggetto.

```
@Check def checkType(Not not) {  
    checkExpectedBoolean(not.expression,  
        ExpressionsPackage.Literals.NOT_EXPRESSION)  
}  
  
@Check def checkType(And and) {  
    checkExpectedBoolean(and.left, ExpressionsPackage.Literals.AND_LEFT)  
    checkExpectedBoolean(and.right, ExpressionsPackage.Literals.AND_RIGHT)  
}  
  
@Check  
def checkType(Or or) {  
    checkExpectedBoolean(or.left, ExpressionsPackage.Literals.OR_LEFT)  
    checkExpectedBoolean(or.right, ExpressionsPackage.Literals.OR_RIGHT)  
}
```

https://github.com/LorenzoBettini/packtpub-xtext-book-2nd-examples_

Il validatore: piccolo esercizio

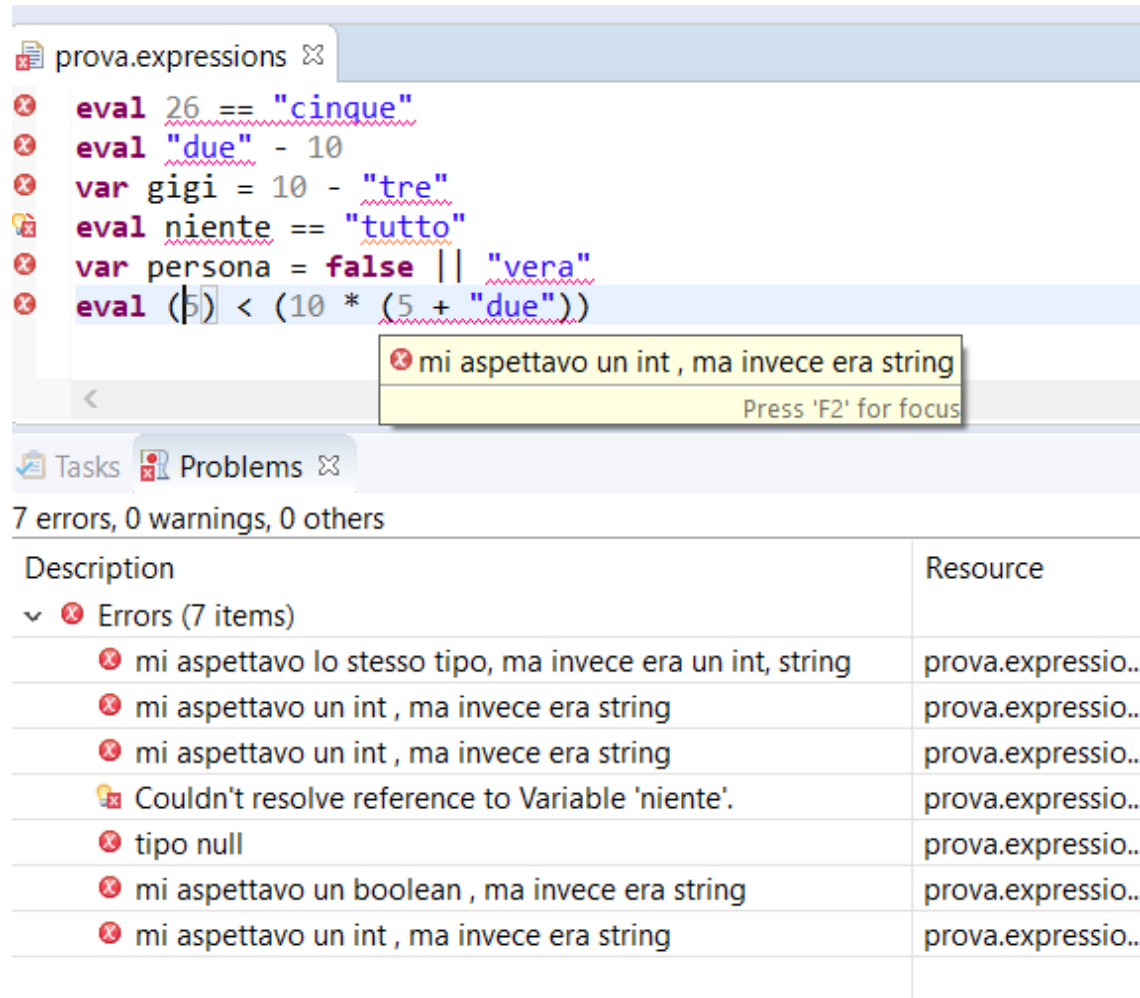
Con lo stesso approccio facciamo (come **esercizio**) il type-checking di Minus e MulOrDiv: partite con questa funzione utility, poi le @Check saranno molto simili a quelle qui sopra:

- checkType(Minus)
- checkType(MulOrDiv)
- Le eReference sempre da ExpressionPackage.Literals (MINUS__LEFT, MINUS__RIGHT, ...)

```
def private checkExpectedInt(Expression exp, EReference reference) {  
    checkExpectedType(exp, ExpressionsTypeComputer.INT_TYPE, reference)  
}
```

Il validatore in azione

Possiamo vedere il nostro primo simpatico validatore in azione 😊



The screenshot shows an IDE window titled 'prova.expressions' containing the following Scala code:

```
eval 26 == "cinque"
eval "due" - 10
var gigi = 10 - "tre"
eval niente == "tutto"
var persona = false || "vera"
eval (5) < (10 * (5 + "due"))
```

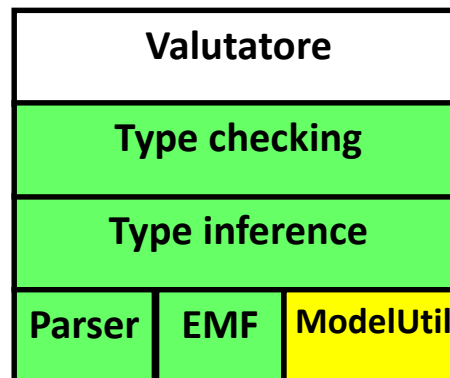
A tooltip for the last line of code displays the error: 'mi aspettavo un int , ma invece era string'.

Below the code editor, the 'Problems' tab is active, showing '7 errors, 0 warnings, 0 others'. The errors are listed in a table:

Description	Resource
✖ Errors (7 items)	
✖ mi aspettavo lo stesso tipo, ma invece era un int, string	prova.expressio...
✖ mi aspettavo un int , ma invece era string	prova.expressio...
✖ mi aspettavo un int , ma invece era string	prova.expressio...
💡 Couldn't resolve reference to Variable 'niente'.	prova.expressio...
✖ tipo null	prova.expressio...
✖ mi aspettavo un boolean , ma invece era string	prova.expressio...
✖ mi aspettavo un int , ma invece era string	prova.expressio...

Parsing dei riferimenti

- ▶ Nel nostro linguaggio abbiamo introdotto anche le **var** come riferimenti a **oggetti già istanziati**. Il nostro linguaggio prevede che i riferimenti siano fatti solamente a oggetti già istanziati (altrimenti si parlerebbe di forward reference).
- ▶ Per gestire i riferimenti nel linguaggio (e quindi anche nel TypeComputer e nel validatore) abbiamo bisogno di programmare la gestione di questi riferimenti.
- ▶ Per fare ciò realizziamo una classe **ExpressionsModelUtil** nel package principale che andrà a gestire i riferimenti alle variabili.



Parsing dei riferimenti

Il seguente algoritmo è solo un esempio di come è possibile gestire i riferimenti, per gestire logiche complesse è poco efficiente.

```
1 package org.example.expressions
2+ import org.example.expressions.expressions.AbstractElement[]
11- class ExpressionsModelUtil {
12-     def isVariableDefinedBefore(VariableRef varRef) {
13         varRef.variablesDefinedBefore.contains(varRef.variable)
14     }
15-     def variablesDefinedBefore(Expression e) {
16         e.getContainerOfType(AbstractElement).variablesDefinedBefore
17     }
18-     def variablesDefinedBefore(AbstractElement containingElement) {
19         val allElements =
20         (containingElement.eContainer as ExpressionsModel).elements
21
22         allElements.subList(0,
23             allElements.indexOf(containingElement)).typeSelect(Variable)
24             .toSet
25     }
26 }
27 }
```

getContainerOfType è un metodo di Eclipse EMF

Scorriamo tutti gli elementi che contengono la nostra expression, definiti prima di essa.

Parsing dei riferimenti: test

Definiamo una nuova clausola di assert apposita per verificare la correttezza del metodo `VariablesDefinedBefore`, quindi definiamo un test:

```
6- def private void assertVariablesDefinedBefore(ExpressionsModel model,
7-     int elemIndex, CharSequence expectedVars
8- ) {
9-     expectedVars.assertEquals(
10-         model.elements.get(elemIndex).variablesDefinedBefore.map[name].join(",")
11-     )
12- }
13- @Test def void variablesBeforeVariable() {
14-     '''
15-         eval true      // (0)
16-         var i = 0       // (1)
17-         eval i + 10     // (2)
18-         var j = i       // (3)
19-         eval i + j      // (4)
20-     '''.parse => [
21-         assertVariablesDefinedBefore(0, "")
22-         assertVariablesDefinedBefore(1, "")
23-         assertVariablesDefinedBefore(2, "i")
24-         assertVariablesDefinedBefore(3, "i")
25-         assertVariablesDefinedBefore(4, "i,j")
26-     ]
27- }
```


Type computer per i riferimenti

La funzione `isVariableDefinedBefore` è utile anche per la type inference dei riferimenti: andiamo a includerla nel `TypeComputer` aggiungendo la `typeFor` anche per i riferimenti a variabile. In questo modo il `TypeComputer` innanzitutto verifica la validità del riferimento, dopodiché restituisce il tipo della variabile a cui il riferimento punta.

```
@Inject extension ExpressionsModelUtil
def dispatch ExpressionsType typeFor(VariableRef varRef) {
  if (!varRef.isVariableDefinedBefore)
    return null
  else {
    return varRef.variable.expression.typeFor
  }
}
```

https://github.com/LorenzoBettini/packtpub-xtext-book-2nd-examples_

Validazione dei riferimenti

Dopo aver testato il funzionamento della `variablesDefinedBefore`, la includiamo nel validatore, con una nuova clausola `Check` che la utilizza per ogni riferimento che incontra:

```
@Inject extension ExpressionsModelUtil
@Check
def void checkForwardReference(VariableRef varRef){
    val variable = varRef.getVariable()
    if(!varRef.isVariableDefinedBefore)
        error("la variabile non è stata dichiarata precedentemente: '"
            +variable.name + "'",
            ExpressionsPackage.eINSTANCE.variableRef_Variable,
            variable.name)
}
```

*prova.expressions

```
var prova = "prova"
```

```
eval prova
```

```
eval prova2
```

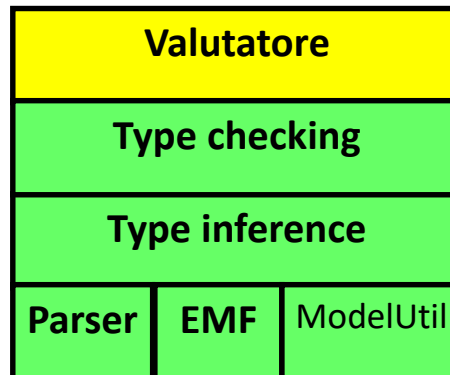
```
var p
```

la variabile non è stata dichiarata precedentemente: 'prova2'

Press 'F2' for focus

Il valutatore

Siamo (finalmente) pronti a scrivere un interprete per il nostro linguaggio. L'idea è che questo interprete riceva gli `AbstractElement` del nostro linguaggio e che li interpreti correttamente come gli oggetti Java corrispondenti. Ad esempio, una espressione booleana deve essere valutata come un oggetto `Boolean` di Java.



Il valutatore

Realizziamo un nuovo package `.interpreter` nel progetto principale, e realizziamo l'`ExpressionsInterpreter`. Esso eseguirà **ricorsivamente** l'interpretazione delle espressioni:

```
}+ import com.google.inject.Inject
}
|- class ExpressionsInterpreter {
}
  @Inject extension ExpressionsTypeComputer
  |- def dispatch Object interpret(Expression e) {
    switch (e) {
    7   IntConstant:
    }     e.value

        BoolConstant:
          Boolean.parseBoolean(e.value)
        StringConstant:
          e.value
        Not:
          !(e.expression.interpret as Boolean)
```

Ci farà molto comodo!

Oggetto Java

(segue..)

Il valutatore

```
MulOrDiv: {  
    val left = e.left.interpret as Integer  
    val right = e.right.interpret as Integer  
    if (e.op == '*')  
        left * right  
    else  
        left / right  
}  
Minus:  
    (e.left.interpret as Integer) - (e.right.interpret as Integer)
```

Per la somma:

In una espressione Plus, se uno dei due operatori è di tipo stringa, allora l'intera espressione è di tipo stringa. Se invece entrambi gli operatori sono di tipo INT, allora l'intera espressione sarà un intero.

Non sono accettate Plus fra valori booleani.

Metodi del Type Computer 😊

```
Plus: {  
    if (e.left.typeFor.isStringType || e.right.typeFor.isStringType)  
        e.left.interpret.toString + e.right.interpret.toString  
    else  
        (e.left.interpret as Integer) + (e.right.interpret as Integer)  
}
```

Concatenazione

Somma

(segue..)

Il valutatore

```
Comparison: {  
  if (e.left.typeFor.isStringType) {  
    val left = e.left.interpret as String  
    val right = e.right.interpret as String  
    switch (e.op) {  
      case '<': left < right  
      case '>': left > right  
      case '>=': left >= right  
      case '<=': left <= right  
      default: false  
    }  
  } else {  
    val left = e.left.interpret as Integer  
    val right = e.right.interpret as Integer  
    switch (e.op) {  
      case '<': left < right  
      case '>': left > right  
      case '>=': left >= right  
      case '<=': left <= right  
      default: false  
    }  
  }  
}
```

Comparison (cast a String o a Integer)

(segue..)

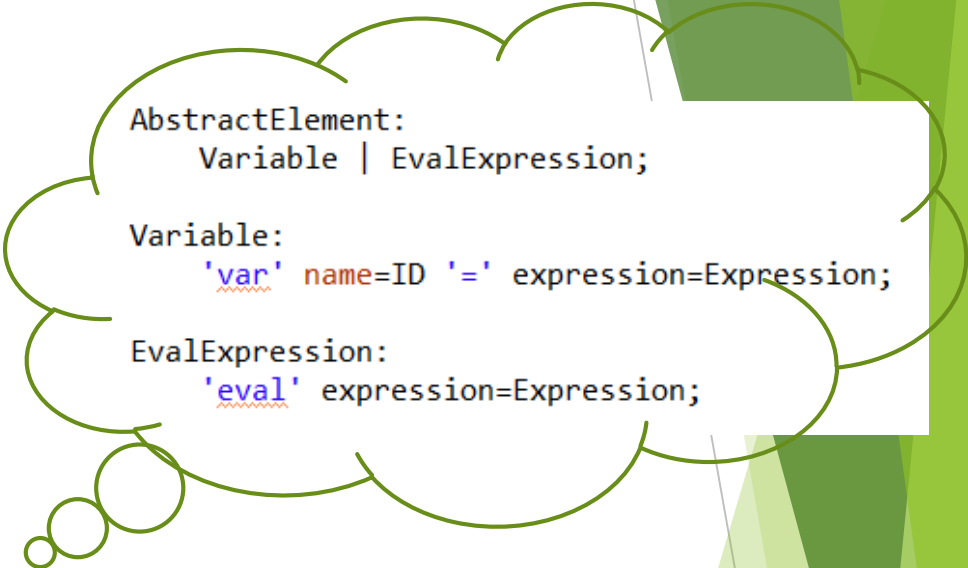
Il valutatore

Per le **var**:

```
,  
VariableRef: {  
  e.variable.interpret  
}
```

Abbiamo raggruppato nella grammatica Variable ed EvalExpression in un oggetto AbstractElement. Usiamo il **dispatch** per eseguire queste valutazioni:

```
def dispatch Object interpret(AbstractElement e) {  
  e.expression.interpret  
}
```



```
AbstractElement:  
  Variable | EvalExpression;  
  
Variable:  
  'var' name=ID '=' expression=Expression;  
  
EvalExpression:  
  'eval' expression=Expression;
```

Usiamo il valutatore

A questo punto abbiamo un valutatore funzionante (il testing è come esercizio, ricordiamoci di aggiungere il nuovo package al Manifest come abbiamo fatto per il Type Computer), ma come possiamo usarlo?

Possiamo, ad esempio, valutare le espressioni e mandare l'output su un file .evaluated. Sul package .generator troviamo la classe

ExpressionsGenerator, dove decidiamo come vogliamo usare l'interprete:

```
import static extension org.eclipse.xtext.nodemodel.util.NodeModelUtils.*
class ExpressionsGenerator extends AbstractGenerator {
    @Inject extension ExpressionsInterpreter

    override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
        resource.allContents.toIterable.filter(ExpressionsModel).forEach [
            fsa.generateFile
                (''«resource.URI.lastSegment».evaluated'',
                 interpretExpressions)
        ]
    }
    def interpretExpressions(ExpressionsModel model) {
        model.elements.map [
            '''«getNode.getTokenText» ~> «interpret»'''
        ].join("\n")
    }
}
```

Nome file

Token interpretato

Questo va a ricostruire completamente il token del DSL a partire dall'AST: è un metodo di NodeModelUtils, classe utility statica che contiene utility sull'AST costruito

Proviamo l'interprete ☺

```
prova.expressions ✕  prova.expressions.evaluated  
  
var gatti = 44  
var fila = 6  
var resto = gatti - (fila * (gatti / fila))  
eval gatti + " gatti in fila per " + fila + " col resto di " + resto  
|
```

```
prova.expressions  prova.expressions.evaluated ✕  
  
var gatti = 44 ~> 44  
var fila = 6 ~> 6  
var resto = gatti - (fila * (gatti / fila)) ~> 2  
eval gatti + " gatti in fila per " + fila + " col resto di " + resto ~> 44 gatti in fila per 6 col resto di 2
```

Funziona!

Un DSL ad oggetti: SmallJava

Tratto da Implementing DSL with Xtext and Xtend, second edition, Lorenzo Bettini, packtpub

La versione completa e ottimizzata si trova su GitHub (il link come riferimento è in calce), nel corso di questa esercitazione costruiremo passo passo una versione semplificata (anche rispetto al libro) di questo esempio.

<https://github.com/LorenzoBettini/packtpub-xtext-book-2nd-examples>

SmallJava

In questo esempio, propedeutico al precedente esempio Expressions DSL, andremo a creare un piccolo DSL orientato agli oggetti, implementando alcuni degli elementi base di un linguaggio ad oggetti, con queste semplificazioni:

- ▶ Le classi non avranno costruttori espliciti
- ▶ Il casting non sarà supportato
- ▶ I tipi base (int, string, bool) non sono considerati
 - ▶ Possiamo aggiungere la grammatica di Expressions, che non andremo a re-implementare in questo esempio
- ▶ Overloading non supportato
- ▶ Ogni classe per accedere ai propri elementi dovrà usare sempre *this*
- ▶ Le variabili devono sempre essere inizializzate
- ▶ Super non è supportata
- ▶ La new non richiede argomenti, dal momento che non esiste il costruttore esplicito
- ▶ I package e gli import non sono supportati

Una estensione di questo linguaggio è disponibile nel Bettini, al capitolo 10 sullo

Grammatica di base

In questo esempio, propedeutico al precedente esempio Expressions DSL, andremo a creare un piccolo DSL orientato agli oggetti, implementando alcuni degli elementi base di un linguaggio ad oggetti:

```
4
5 SJProgram:
6     classes += SJClass*
7 ;
8
9 SJClass: 'class' name=ID ('extends' superclass=[SJClass])?
10     '{' members += SJMember* '}'
11 ;
12
13 SJMember:
14     SJField | SJMethod
15 ;
16
17 fragment SJTypedDeclaration *:
18     type=[SJClass] name=ID
19 ;
```

Eventuale superclasse

membri

Altro zucchero sintattico di Xtext: con la regola di parsing **fragment** introduciamo un frammento di regola che riutilizziamo più volte.

Con l'asterisco * indichiamo che non vogliamo creare un nuovo tipo nell'AST, dato che andremo noi a creare i vari sottotipi che desideriamo (che sono diversi da SJTypedDeclaration).

Grammatica di base

Introduciamo campi dell'oggetto, metodi, parametri del metodo e corpo del metodo:

```
--
21 SJField:
22     SJTypedDeclaration ';'
23 ;
24
25 SJMethod:
26     SJTypedDeclaration
27     '(' (params+=SJParameter
28         (',' params +=SJParameter)*
29         )?
30     ')' body = SJBlock
31 ;
32
33 SJParameter:
34     SJTypedDeclaration
35 ;
36
37 SJBlock:
38     {SJBlock} '{' statements += SJStatement* '}'
39 ;
40
```

Campo dell'oggetto

Metodo

Parametro

Eventuali altri parametri

Corpo del metodo

Grammatica di base

Introduciamo le istruzioni del linguaggio, la return e le dichiarazioni di variabili. In particolare, una istruzione può essere una dichiarazione, una return, una espressione o un if.

```
» SJStatement:  
    SJVariableDeclaration |  
    SJReturn |  
    SJExpression ';' |  
    SJIfStatement  
;  
  
» SJReturn:  
    'return' expression = SJExpression ';' ;  
  
» SJVariableDeclaration:  
    SJTypedDeclaration '=' expression= SJExpression ';' ;
```

Dangling Else Problem

Scriviamo la grammatica per la if-then-else:

```
51
52 SJIfStatement:
53     'if' '(' expression=SJExpression ')' thenBlock=SJIfBlock
54     ('else' elseBlock=SJIfBlock)?
55 ;
56
57 SJIfBlock returns SJBlock:
58     statements+=SJStatement
59     | SJBlock;
```

Singola istruzione
o più istruzioni con le parentesi graffe

Tuttavia, se la scriviamo in questo modo, ci ritroveremo in seguito, quando andremo a realizzare gli Xtext artifacts, con questo errore da parte di ANTLR:

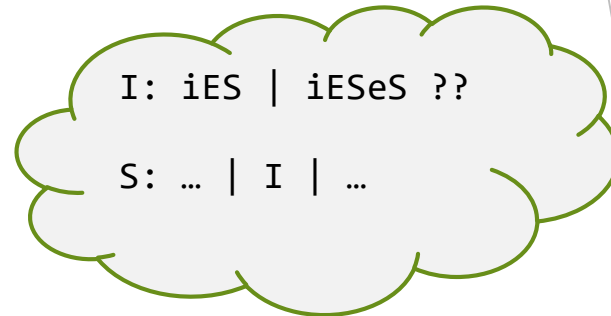
Mai ignorare i warning di ANTLR!

```
warning(200): ../org.xtext.example.smalljava/src-
gen/org/xtext/example/smalljava/parser/antlr/internal/InternalSmallJava.g:751:3:
Decision can match input such as "'else'" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
warning(200): ../org.xtext.example.smalljava.ide/src-
gen/org/xtext/example/smalljava/ide/contentassist/antlr/internal/InternalSmallJava.g:1809:2:
Decision can match input such as "'else'" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
```

Dangling Else Problem

Si tratta di un *vecchio classico sempreverde* (cit. ED), denominato **Dangling Else Problem**: la grammatica è ambigua. Vediamo questo esempio:

```
if(...)
  if(...)
    istruzione
  else
    istruzione
```



Il parser non è in grado di stabilire a quale dei due if appartiene l'else, qui si nota solo grazie alla tabulazione, e per come siamo abituati ad usare le if-else.

Predicati sintattici in Xtext

Per risolvere questo problema usiamo un'altra importante caratteristica di Xtext, che sono i **predicati sintattici**:

```
52 SJIfStatement:  
53     'if' '(' expression=SJExpression ')' thenBlock=SJIfBlock  
54     (=> 'else' elseBlock=SJIfBlock)?  
55 ;
```

Questo è un predicato sintattico

Con il predicato sintattico indichiamo al parser che, nel momento in cui incontra la parola chiave `else`, legherà direttamente le dichiarazioni (la parte fra parentesi) all'ultimo `if` dichiarato. Questo è il comportamento tipico dei linguaggi con la `if`.

Un'alternativa (che non usa i predicatori sintattici) è usare il backtracking con il parser ANTLR, tuttavia è caldamente sconsigliata, dato che è solo fonte di ulteriori problemi difficilmente individuabili..

Superclassi di oggetti Xtext

Le due regole di seguito servono per forzare la creazione di di superclassi comuni:

- ▶ I parametri e le dichiarazioni di variabili
- ▶ I classi, i metodi, gli oggetti membri delle classi, le variabili e i parametri

```
⇒ SJSymbol:  
    SJVariableDeclaration | SJParameter;  
  
⇒ SJNamedElement:  
    SJClass | SJMember | SJSymbol  
;
```

Queste superclassi ci faranno comodo in fase di validazione. L'IDE ce le indica in grigio dal momento in cui queste regole non verranno mai chiamate dal parser.

Assegnamenti su SmallJava

Scriviamo la grammatica per le espressioni, in questo esempio non tratteremo tutte quelle viste nell'esempio Expressions, ci limiteremo agli assegnamenti e alle chiamate dei metodi, gestendo la ricorsione nel modo corretto.

- Per gli assegnamenti, abbiamo che $a = b = c$ dovrebbe essere letto dal parser come $a = (b = c)$ quindi abbiamo la **ricorsione a destra**

```
SJAssignment returns SJExpression:  
  SJSelectionExpression  
  ({SJAssignment.left=current} '=' right=SJExpression ?;)
```

Ricorsione a destra

Espressioni su SmallJava

- ▶ Per la chiamata di campi o metodi di un oggetto, dovremmo riuscire a gestire chiamate del tipo `a.b().c.d()`
- ▶ La distinzione fra chiamata a metodo e selezione un campo è memorizzata sulla feature booleana *methodinvocation*
- ▶ La parte **sinistra** di una chiamata è il **receiver**
 - ▶ È obbligatoria

• SJSelectionExpression **returns** SJExpression:

```
SJTerminalExpression
```

```
(
```

```
{SJMemberSelection.receiver=current} '.'
```

```
member=[SJMember]
```

```
(methodinvocation?='('
```

```
(args+=SJExpression (',' args+=SJExpression)*)? ')'
```

```
)?
```

```
)*;
```

Il receiver

Un SJField o un SJMethod

La feature *methodinvocation* è *true* se ci sono le parentesi

Se è un metodo contiene anche questa parte. Il metodo può contenere eventualmente parametri.

Loose grammar, strict validation

La grammatica che abbiamo appena definito per le chiamate non fa distinzione fra la selezione di field dell'oggetto o le chiamate ai metodi di quell'oggetto. Potremmo pensare a una versione della grammatica che esplicita nell'AST quale delle due operazioni sono state effettuate:

```
SJTerminalExpression  
(  
  ({SJMethodInvocation.receiver=current} '.'  
    method=[SJMethod]  
    '(' (args+=SJExpression(',') args+=SJExpression)*? ')'  
  ) |  
  ({SJFieldSelection.receiver=current} '.' field = [SJField])  
)*  
;
```

Se abbiamo una invocazione creiamo un SJMethodInvocation

NON useremo
questo pezzo di
grammatica!!

Se stiamo selezionando un campo dell'oggetto abbiamo un
SJFieldSelection

Tuttavia questa "espressività extra" offerta da questa grammatica darà un ulteriore carico al parser e alcune funzioni dell'IDE come il **content-assist** (per suggerire un campo dell'oggetto o una sua funzione) non funzioneranno come previsto.

Quindi un buon approccio generale quando si definisce un linguaggio in Xtext è di seguire il principio "**loose grammar, strict validation**" [Zarnkow 2012] ovvero **delegare** al validatore il compito di distinguere fra i sottotipi che è possibile invocare.

Terminali

Infine abbiamo tutti i terminali:

- Espressioni fra parentesi
- New (solo senza argomenti)
- This (da usare sempre per fare riferimento a propri campi o funzioni)
 - Questo perché la SJSymbolRef fa riferimento a un SJSymbol. Se facessimo riferimento a un SJMember non ci sarebbe bisogno della this, tuttavia ci sarebbe bisogno di un motore di scoping (che al momento non affrontiamo) per far funzionare il validatore e il valutatore.
- Null
- Booleani

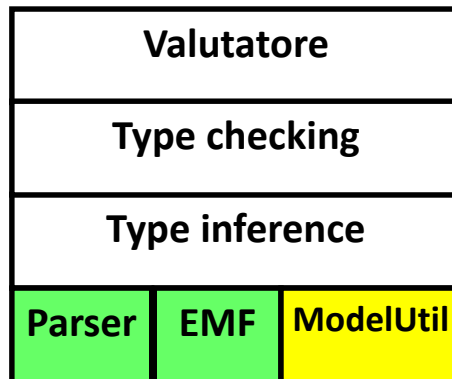
SJTerminalExpression **returns** SJExpression:

```
{SJThis} 'this' |  
{SJNull} 'null' |  
{BoolConstant} value=('true' | 'false') |  
{SJSymbolRef} symbol=[SJSymbol] |  
{SJNew} 'new' type=[SJClass] '(' ' ' ')' |  
'(' SJExpression ')';
```

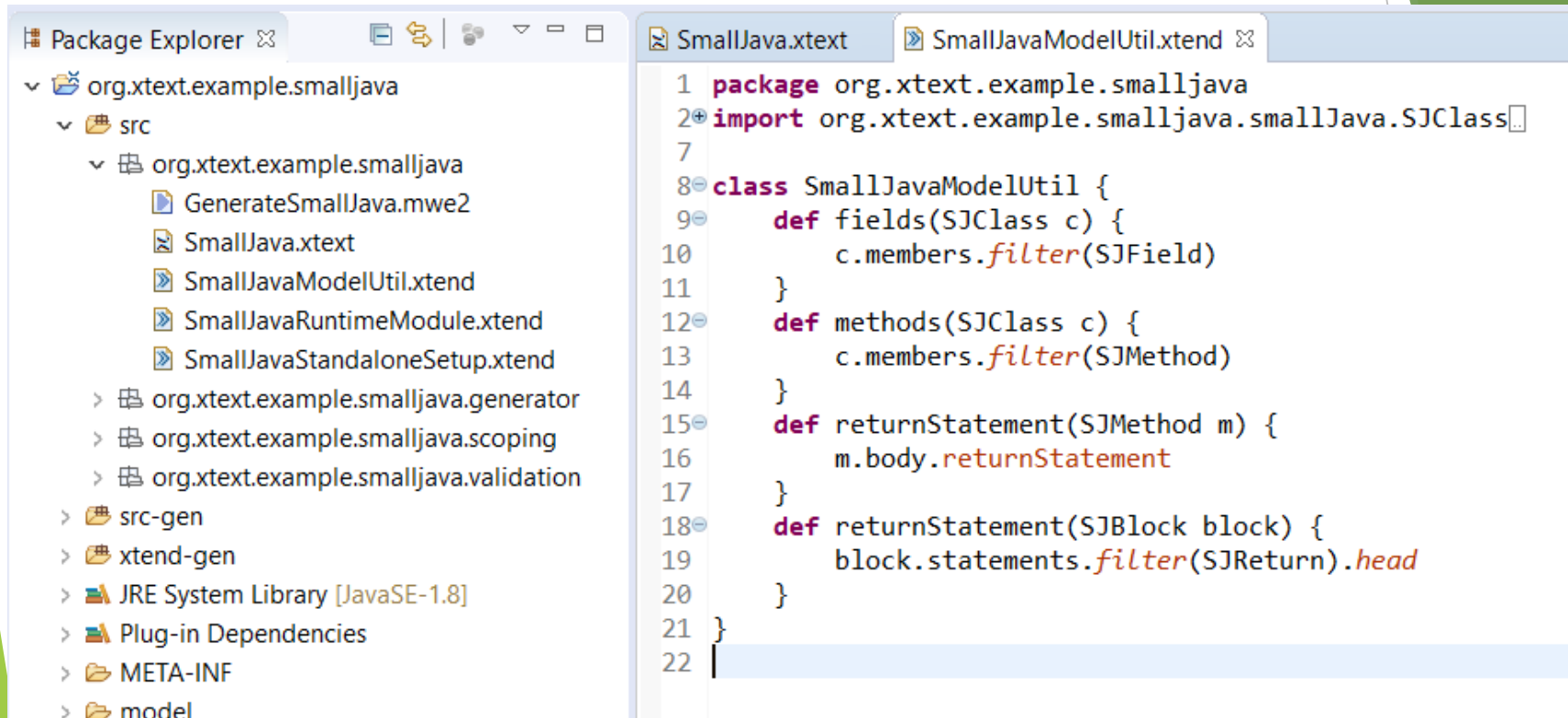
Metodi utility

Definiamo, in una nuova classe Xtext, dei metodi utility per il modello, in particolare:

- Un metodo per ottenere i **campi** di una **classe** SmallJava
- Un metodo per ottenere i **metodi** di una **classe** SJ
- Un metodo per ottenere la **return** di un **metodo** SJ
- Un metodo per ottenere la **return** da un **blocco** di codice SJ



Metodi utility



Package Explorer

- org.xtext.example.smalljava
 - src
 - org.xtext.example.smalljava
 - GenerateSmallJava.mwe2
 - SmallJava.xtext
 - SmallJavaModelUtil.xtend
 - SmallJavaRuntimeModule.xtend
 - SmallJavaStandaloneSetup.xtend
 - org.xtext.example.smalljava.generator
 - org.xtext.example.smalljava.scoping
 - org.xtext.example.smalljava.validation
 - src-gen
 - xtend-gen
 - JRE System Library [JavaSE-1.8]
 - Plug-in Dependencies
 - META-INF
 - model

SmallJava.xtext

SmallJavaModelUtil.xtend

```
1 package org.xtext.example.smalljava
2+ import org.xtext.example.smalljava.smallJava.SJClass
7
8= class SmallJavaModelUtil {
9=   def fields(SJClass c) {
10       c.members.filter(SJField)
11   }
12=   def methods(SJClass c) {
13       c.members.filter(SJMethod)
14   }
15=   def returnStatement(SJMethod m) {
16       m.body.returnStatement
17   }
18=   def returnStatement(SJBlock block) {
19       block.statements.filter(SJReturn).head
20   }
21 }
22
```

<https://github.com/LorenzoBettini/packtpub-xttext-book-2nd-examples>

Testing del linguaggio

Per il testing vedremo solo due casi notevoli:

- Associatività degli operatori
- Dangling else

```
class SmallJavaParsingTest {  
  @Inject extension ParseHelper<SJProgram>  
  @Inject extension SmallJavaModelUtil  
  
  def private assertAssociativity(SJStatement s, CharSequence expected) {  
    expected.toString.assertEquals(s.stringRepr)  
  }  
  
  def private String stringRepr(SJStatement s) {  
    switch (s) {  
      SJAssignment: '''(«s.left.stringRepr» = «s.right.stringRepr）」'''  
      SJMemberSelection: '''(«s.receiver.stringRepr».«s.member.name）」'''  
      SJThis: "this"  
      SJNew: '''new «s.type.name»()'''  
      SJNull: "null"  
      SJSymbolRef: s.symbol.name  
      SJReturn: s.expression.stringRepr  
    }  
  }  
}
```

Classi utility per testare la
associatività (molto simili a
quelle di Expressions)

Rappresentazione verbosa

Testing del linguaggio

- Associatività degli operatori

```
@Test def void testMemberSelectionLeftAssociativity() {  
    ...  
    class A {  
        A m() { return this.m().m(); }  
    }  
    ...  
    .parse.classes.head.methods.head.  
    body.statements.last.  
    assertAssociativity("((this.m).m)")  
}
```

Come pensavate di fare senza la dependency injection? 😊

```
@Test def void testAssignmentRightAssociativity() {  
    ...  
    class A {  
        A m() {  
            A f = null;  
            A g = null;  
            f = g = null;  
        }  
    }  
    ...  
    .parse.classes.head.methods.head.  
    body.statements.last.  
    assertAssociativity("(f = (g = null))")  
}
```

Testing del linguaggio

- Dangling else

```
@Test def void testElse() {  
    ...
```

```
class C {  
    C c;  
    C m() {  
        if (true)  
            if (false)  
                this.c = null;  
        else  
            this.c = null;  
        return this.c;  
    }  
}
```

Come esercizio proviamo a vedere se funziona correttamente anche con le graffe!

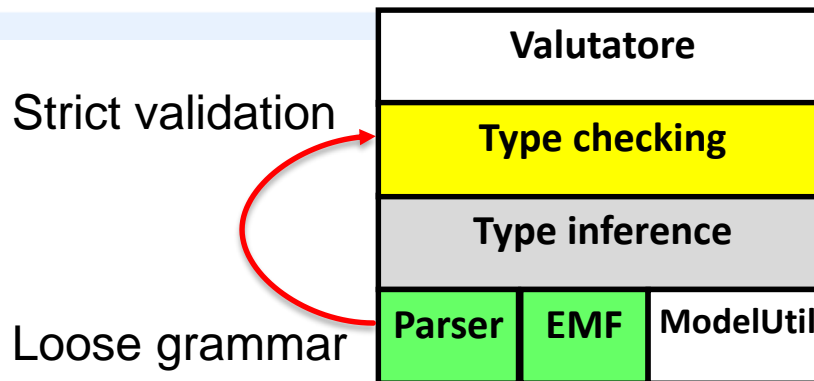
```
... .parse => [  
    val ifS = (classes.head.methods.head.  
        body.statements.head as SIfStatement)  
    ifS	elseBlock.assertNull  
    // thus the else is associated to the inner if  
]  
}
```

Metodo di SIfStatement (generato da Xtext)

Validazione: metodi vs attributi

Abbiamo delegato la distinzione fra chiamate a metodo e accesso ad attributi di un oggetto al validatore (principio loose grammar, strict validation). Aggiungiamo quindi la regola al validatore:

```
public static val SELEZIONE_ATTRIBUTO_IN_CHIAMATA = ISSUE_CODE_PREFIX + "SelezioneAttributoInChiamata"
public static val CHIAMATA_METODO_IN_ATTRIBUTO = ISSUE_CODE_PREFIX + "ChiamataMetodoInAttributo"
@Check def void checkSelezioneAttributo(SJMemberSelection sel){
    val membro = sel.member
    if(membro instanceof SJField && sel.methodinvocation)
        error("Hai usato l'attributo '"+ membro.name + "' come se fosse un metodo!",
            SmallJavaPackage.eINSTANCE.SJMemberSelection_Methodinvocation,
            SELEZIONE_ATTRIBUTO_IN_CHIAMATA)
    else if(membro instanceof SJMethod && !sel.methodinvocation)
        error("Hai usato il metodo '"+ membro.name + "' come se fosse un attributo!",
            SmallJavaPackage.eINSTANCE.SJMemberSelection_Methodinvocation,
            CHIAMATA_METODO_IN_ATTRIBUTO)
}
```



Validazione: metodi vs attributi

```
class Prova {  
    Prova attributo;  
    Prova metodo(){  
        Prova var = this.attributo();  
        var = this.metodo; }  
}
```

Tasks Problems

2 errors, 0 warnings, 0 others

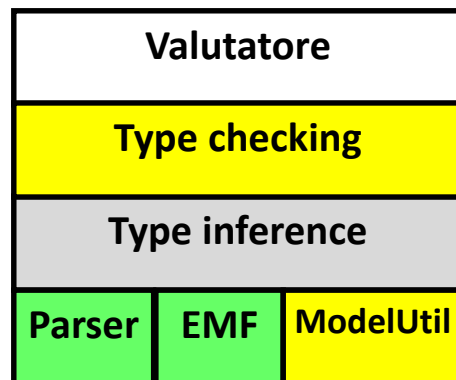
Description

Errors (2 items)

- Hai usato il metodo 'metodo' come se fosse un attributo!
- Hai usato l'attributo 'attributo' come se fosse un metodo!

Validazione: ereditarietà cicliche

Al momento il nostro parser non è in grado di individuare (e segnalare) ereditarietà cicliche. Inizialmente dobbiamo realizzare una funzione utility (da aggiungere al **ModelUtil**) che esplora l'albero di ereditarietà, e si ferma se raggiunge la radice o se incontra un loop:



Per questo tipo di controllo
non è necessaria!

Validazione: ereditarietà cicliche

Al momento il nostro parser non è in grado di individuare (e segnalare) ereditarietà cicliche. Inizialmente dobbiamo realizzare una funzione utility (da aggiungere al **ModelUtil**) che esplora l'albero di ereditarietà, e si ferma se raggiunge la radice o se incontra un loop:

```
def classHierarchy(SJClass c) {  
    val visited = newLinkedHashSet()  
  
    var current = c.superclass  
    while (current != null && !visited.contains(current)) {  
        visited.add(current)  
        current = current.superclass  
    }  
    visited  
}
```

Funzione della classe Xtend CollectionLiterals

Zucchero sintattico: la return si può omettere!

Validazione: ereditarietà cicliche

Mettiamo quindi mano al Validator per aggiungere la clausola che segnala la gerarchia ciclica:

```
class SmallJavaValidator extends AbstractSmallJavaValidator {  
  
    protected static val ISSUE_CODE_PREFIX = "org.example.smalljava.";  
    public static val GERARCHIA_CICLICA = ISSUE_CODE_PREFIX + "GerarchiaCiclica";  
  
    @Inject extension SmallJavaModelUtil  
  
    @Check def checkGerarchiaClassi(SJClass c){  
        if(c.classHierarchy.contains(c)){  
            error("ciclo nella gerarchia della classe '" + c.name + "'",  
                SmallJavaPackage.eINSTANCE.SJClass_Superclass,  
                GERARCHIA_CICLICA, c.superclass.name)  
        }  
    }  
}
```

main.smalljava

```
class A extends C {}  
class B extends A {}  
class C extends B {}
```

ciclo nella gerarchia della classe 'C'

Press 'F2' for focus

Unreachable code

Potrei saltarlo, caso poco interessante.

Controllo dei duplicati avanzato

Normalmente viene usato il validatore di default *NamesAreUniqueValidator*. Lbettini propone di utilizzare una struttura dati speciale di Google chiamata HashMultimap, che funziona in maniera molto efficiente, per evitare la complessità quadratica di ricerca dei duplicati che si incontrerebbe in un programma scritto con questo linguaggio.

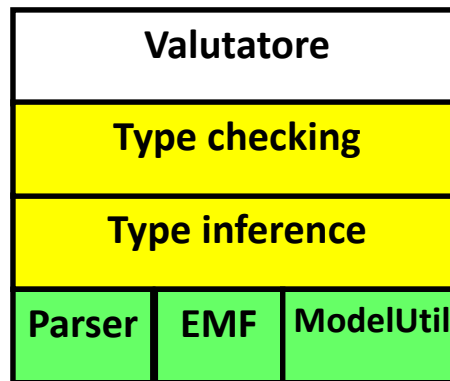
Tuttavia, per scopo didattico, salterei questa parte per il momento.

Type computer

Come per il precedente esempio Expressions, andiamo a separare la parte di inferenza dei tipi dal controllo della loro corrispondenza. La type inference per SmallJava è più semplice di quella di Expressions, dal momento che:

- tutti i tipi afferiscono a SJClass, tranne null

Realizziamo quindi la prima parte di SmallJavaTypeComputer nel nuovo sottopackage typing.



Type computer

```
import static extension org.eclipse.xtext.EcoreUtil2.*
class SmallJavaTypeComputer {
  def SJClass typeFor(SJExpression e) {
    switch (e) {
      SJNew: e.type
      SJSymbolRef: e.symbol.type
      SJMemberSelection: e.member.type
      SJAssignment: e.left.typeFor
      SJThis: e.getContainerOfType(SJClass)
    }
  }
}
```

- Il tipo di una classe che stiamo istanziando è (ovviamente) il tipo che gli stiamo dando
- Il tipo una variabile o di un parametro è il tipo della classe con cui è costruito
- Il tipo di un riferimento è il tipo dell'attributo o del metodo a cui si fa riferimento
- Il tipo di un assegnamento è il tipo dell'oggetto che viene assegnato
- Il tipo di *this* è semplicemente il tipo della classe stessa.

Type computer

In SmallJava abbiamo anche definito dei terminali: boolean e null. Dato che per definire il tipo di un oggetto abbiamo bisogno di una SJClass, andiamo a realizzare una istanza statica di SJClass per ognuno di questi terminali:

```
private static val factory = SmallJavaFactory.eINSTANCE
public static val BOOLEAN_TYPE = factory.createSJClass => [name = 'booleanType']
public static val NULL_TYPE = factory.createSJClass => [name = 'nullType']
```

Dopodichè aggiungiamo i tipi alla typeFor:

```
def SJClass typeFor(SJExpression e) {
  switch (e) {
    SJNew: e.type
    SJSymbolRef: e.symbol.type
    SJMemberSelection: e.member.type
    SJAssignment: e.left.typeFor
    SJThis: e.getContainerOfType(SJClass)
    SJNull: NULL_TYPE
    BoolConstant: BOOLEAN_TYPE
  }
}
```

Type conformance

In un linguaggio ad oggetti dobbiamo controllare anche la correttezza dei tipi delle classi, ovvero se stiamo utilizzando la corretta gerarchia di classi. Ad esempio, scrivendo

```
Tubero t = new Patata();
```

È necessario che Patata sia stata definita come

```
Class Patata extends Tubero { ... }
```

Andiamo a quindi verificare la type conformance in una **classe a parte**, **SmallJavaTypeConformance**.

Valutatore		
Type checking		
Type conformance	Type inference	
Parser	EMF	ModelUtil

Type conformance

Consideriamo le seguenti regole:

- Il null è conforme a ogni tipo: può essere associato a ogni variabile, attributo o passato come parametro
- Una classe non è sottoclasse di sé stessa, ma è conforme a sé stessa

```
import static org.xtext.example.smalljava.typing.SmallJavaTypeComputer.*
class SmallJavaTypeConformance {
    @Inject extension SmallJavaModelUtil

    def isConformant(SJClass c1, SJClass c2) {
        c1 == NULL_TYPE || // null can be assigned to everything
        c1 == c2 ||
        c1.isSubclassOf(c2)
    }

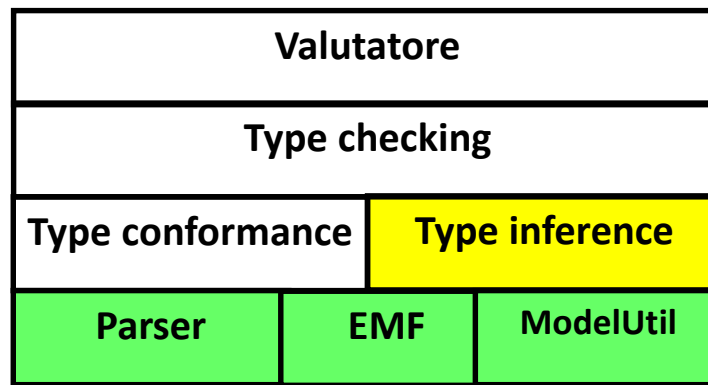
    def isSubclassOf(SJClass c1, SJClass c2) {
        c1.classHierarchy.contains(c2)
    }
}
```

Test di esempio, fatene di più complessi!

```
@Inject extension SmallJavaTypeConformance
@Test def void testClassConformance(){
    ...
    class A{}
    ''''.parse
    .classes => [
        get(0).isConformant(get(0)).assertTrue
    ]
}
```

Computazione del tipo previsto

- Per ogni **contesto di utilizzo** il validatore si deve aspettare dei tipi differenti. Ad esempio, se stiamo validando un if, ci aspettiamo che il tipo dell'espressione all'interno di un if sia di tipo booleano.
- Invece di implementare una lunga serie di clausole @Check, possiamo separare la **computazione del tipo atteso** e poi nel validatore **verificare** se il tipo previsto corrisponde al tipo effettivo.
- Quindi nel Type Computer aggiungiamo una funzione `expectedType(SJExpression)` che per ogni espressione determina il tipo previsto.



Computazione del tipo previsto

```
static val ep = SmallJavaPackage.eINSTANCE
def expectedType(SJExpression e) {
  val c = e.eContainer
  val f = e.eContainingFeature
  switch (c) {
    SJVariableDeclaration:
      c.type
    SJAssignment case f == ep.SJAssignment_Right:
      typeFor(c.left)
    SJReturn:
      c.getContainerOfType(SJMethod).type
    case f == ep.SJIfStatement_Expression:
      BOOLEAN_TYPE
    SJMemberSelection case f == ep.SJMemberSelection_Args: {
      try {
        (c.member as SJMethod).params.get(c.args.indexOf(e)).type
      } catch (Throwable t) {
        null
      }
    }
  }
}
```

Per SmallJava dobbiamo tenere conto di queste regole:

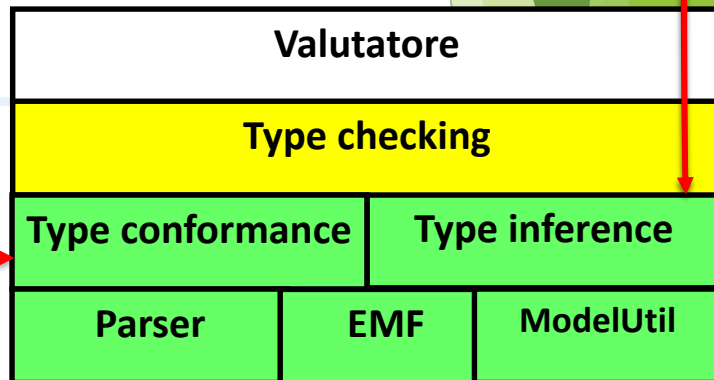
1. Se stiamo inizializzando una **variabile** di tipo T, ci aspettiamo una espressione di tipo T
2. Se stiamo effettuando un **assegnamento**, ci aspettiamo di avere alla destra un oggetto dello stesso tipo
3. Per la **return** ci aspettiamo lo stesso tipo del metodo
4. Per un if ci aspettiamo una espressione booleana
5. Come tipi degli **argomenti** di una invocazione a metodo ci aspettiamo i tipi dichiarati

Se non vi erano parametri il ParsingHelper di Xtext lancia una exception, che catturiamo (alla buona)

Type checking

A questo punto aggiungiamo la clausola @Check nel validatore per verificare se il tipo atteso coincide con il tipo effettivo:

```
@Inject extension SmallJavaTypeComputer
@Inject extension SmallJavaTypeConformance
public static val TIPI_INCOMPATIBILI = ISSUE_CODE_PREFIX + "TipiIncompatibili"
@Check def void checkConformance(SJExpression exp) {
  val actualType = exp.typeFor
  val expectedType = exp.expectedType
  if (expectedType === null || actualType === null)
    return; // nothing to check
  if (!actualType.isConformant(expectedType)) {
    error("Tipi incompatibili. Mi aspettavo un '" + expectedType.name
      + "' ma invece era '" + actualType.name + "'",
      null, TIPI_INCOMPATIBILI);
  }
}
```



Type checking

```
class Classina {}  
class Classe {  
    Classina campo;  
    Classe funzione(Class parametro){  
        if(true) return new Classe();  
        else if(this.campo)  
            return this.campo;  
    }  
}
```

Tasks Problems

2 errors, 0 warnings, 0 others

Description

Errors (2 items)

- Tipi incompatibili. Mi aspettavo un 'booleanType' ma invece era 'Classina'
- Tipi incompatibili. Mi aspettavo un 'Classe' ma invece era 'Classina'

Type conformance: override

Un'ultima cosa che dobbiamo controllare è la correttezza degli override:

- Il tipo di ritorno deve essere lo stesso del metodo di origine
- I parametri devono essere dello stesso tipo

Per implementare questo controllo nel validatore usiamo il metodo `classHierarchy()` che abbiamo implementato nel `ModelUtil`. Inoltre [Bettini] consiglia di usare delle utility per le mappe di Xtext che sfruttano le lambda expression per individuare i riferimenti alla gerarchia: nel `ModelUtil` andiamo a creare una nuova funzione che ritorna una **mappa invertita** dei metodi della gerarchia, in modo di poter risalire rapidamente la gerarchia dei metodi che è stata costruita:

```
def classHierarchyMethods(SJClass c) {  
  c.classHierarchy.toList.reverseView.  
    map[methods].flatten.toMap[name]  
}
```

Ribaltiamo la gerarchia delle classi..

..estrai-
am
o una lista
di tutti i
metodi..

..la
trasformiam
o in un
unico
iteratore..

..infine con toMap creiamo una
mappa di elementi indicizzati per
nome

Type conformance: override

I metodi *map* e *toMap* di Xtext utilizzano la programmazione funzionale per classificare dinamicamente gli elementi dell'AST!

```
● <SJClass, Iterable<SJMethod>> List<Iterable<SJMethod>> ListExtensions.map(List<SJClass> original,  
Function1<? super SJClass, ? extends Iterable<SJMethod>> transformation)
```

Returns a list that performs the given `transformation` for each element of `original` when requested. The mapping is done lazily. That is, subsequent iterations of the elements in the list will repeatedly apply the transformation. The returned list is a transformed view of `original`; changes to `original` will be reflected in the returned list and vice versa (e.g. invocations of [List.remove\(int\)](#)).

```
● <String, SJMethod> Map<String, SJMethod> IterableExtensions.toMap(Iterable<? extends SJMethod> value  
Function1<? super SJMethod, String> computeKeys)
```

Returns a map for which the [Map.values](#) are the given elements in the given order, and each key is the product of invoking a supplied function `computeKeys` on its corresponding value. If the function produces the same key for different values, the last one will be contained in the map.

Type Parameters:

<K>

<V>

Parameters:

values the values to use when constructing the `Map`. May not be `null`.

computeKeys the function used to produce the key for each value. May not be `null`.

Returns:

a map mapping the result of evaluating the function `keyFunction` on each value in the input collection to that value

```
def classHierarchyMethods(SJClass c) {  
    c.classHierarchy.toList.reverseView.  
        map[methods].flatten.toMap[name]  
}
```

Type conformance: override

Dopodichè aggiungiamo la clausola @Check nel validatore per verificare il subtyping del metodo e dei parametri nel validatore:

```
public static val OVERRIDE_SBAGLIATO = ISSUE_CODE_PREFIX + "OverrideSbagliato"
```

```
@Check def void checkMethodOverride(SJClass c) {  
  val hierarchyMethods = c.classHierarchyMethods  
  for (m : c.methods) {  
    val overridden = hierarchyMethods.get(m.name)  
    if (overridden != null &&  
        (!m.type.isConformant(overridden.type) ||  
         !m.params.map[type].elementsEqual(overridden.params.map[type]))) {  
      error("Il metodo '" + m.name + "' deve effettuare l'override in modo corretto!",  
            m, SmallJavaPackage.eINSTANCE.SJNamedElement_Name,  
            OVERRIDE_SBAGLIATO)  
    }  
  }  
}
```

Da SmallJavaTypeConformance

```
class Classe {  
  Classe funzione(Class parametro){  
    return new Classe();  
  }  
}  
class ClasseNuova extends Classe {  
  Classe funzione(ClassNuova parametro){  
    ret  
  }  
}
```

Il metodo 'funzione' deve effettuare l'override in modo corretto!

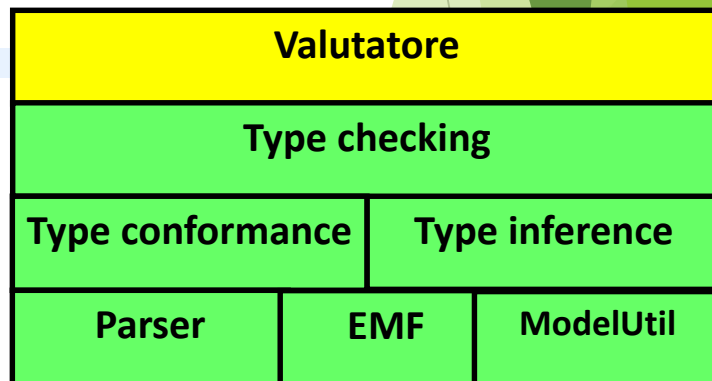
Press 'F2' for focus

Compilatore

Dopo aver realizzato queste (prime) clausole del validatore ci fermiamo per realizzare un primo valutatore che da un sorgente SmallJava genererà un sorgente Java, tralasciando lo scoping (le clausole public, private e quant'altro). Mettiamo quindi mano allo **SmallJavaGenerator**, scrivendo i metodi per compilare ricorsivamente il nostro linguaggio:

- **compileTypeReference** per estrarre il nome di una SJClass

```
@Inject extension IQualifiedNameProvider
def compileTypeReference(SJClass c) {
  c.fullyQualifiedName.toString
}
```



Compilatore

Mettiamo quindi mano allo **SmallJavaGenerator**, scrivendo i metodi per compilare ricorsivamente il nostro linguaggio :

- **compileExpression** per compilare una espressione

```
def String compileExpression(SJStatement s) {  
  switch (s) {  
    SJStringConstant: ''' + s.value + '''  
    SJIntConstant: s.value.toString  
    SJBoolConstant: s.value  
    SJNull: "null"  
    SJThis: "this"  
    SJSymbolRef: s.symbol.name  
    SJNew: "new " + s.type.compileTypeReference + "()  
    SJAssignment: {  
      s.left.compileExpression + " = " + s.right.compileExpression  
    }  
    SJMemberSelection: {  
      s.receiver.compileExpression + "." + s.member.name +  
      if (s.methodinvocation) {  
        "(" + s.args.map[compileExpression].join(", ") + ")"  
      } else {  
        ""  
      }  
    }  
  }  
}
```


Compilatore

Mettiamo quindi mano allo **SmallJavaGenerator**, scrivendo i metodi per compilare ricorsivamente il nostro linguaggio :

- **compileStatement** per compilare if, return, e dichiarazioni di variabili

```
def String compileStatement(SJStatement s) {  
  switch (s) {  
    SJVariableDeclaration: '''«s.type.compileTypeReference» «s.name»  
      = «s.expression.compileExpression» ;'''  
    SJReturn: "return " + s.expression.compileExpression + ";"  
    SJIfStatement: '''  
      if («s.expression.compileExpression»)  
        «s.thenBlock.compileBlock»  
      «IF s.elseBlock != null»  
      else  
        «s.elseBlock.compileBlock»  
      «ENDIF»  
      '''  
    default: s.compileExpression + ";"  
  }  
}
```

Compilatore

Mettiamo quindi mano allo **SmallJavaGenerator**, scrivendo i metodi per compilare ricorsivamente il nostro linguaggio :

- **compileParam** per compilare i parametri
- **compileBlock** per iterare su un blocco di codice

```
def compileParam(SJParameter p) {  
    '''«p.type.compileTypeReference» «p.name»'''  
}
```

```
def compileBlock(SJBlock block) '''  
    {  
        «FOR s : block.statements»  
        «compileStatement(s)»  
        «ENDFOR»  
    }  
'''
```

Generatore

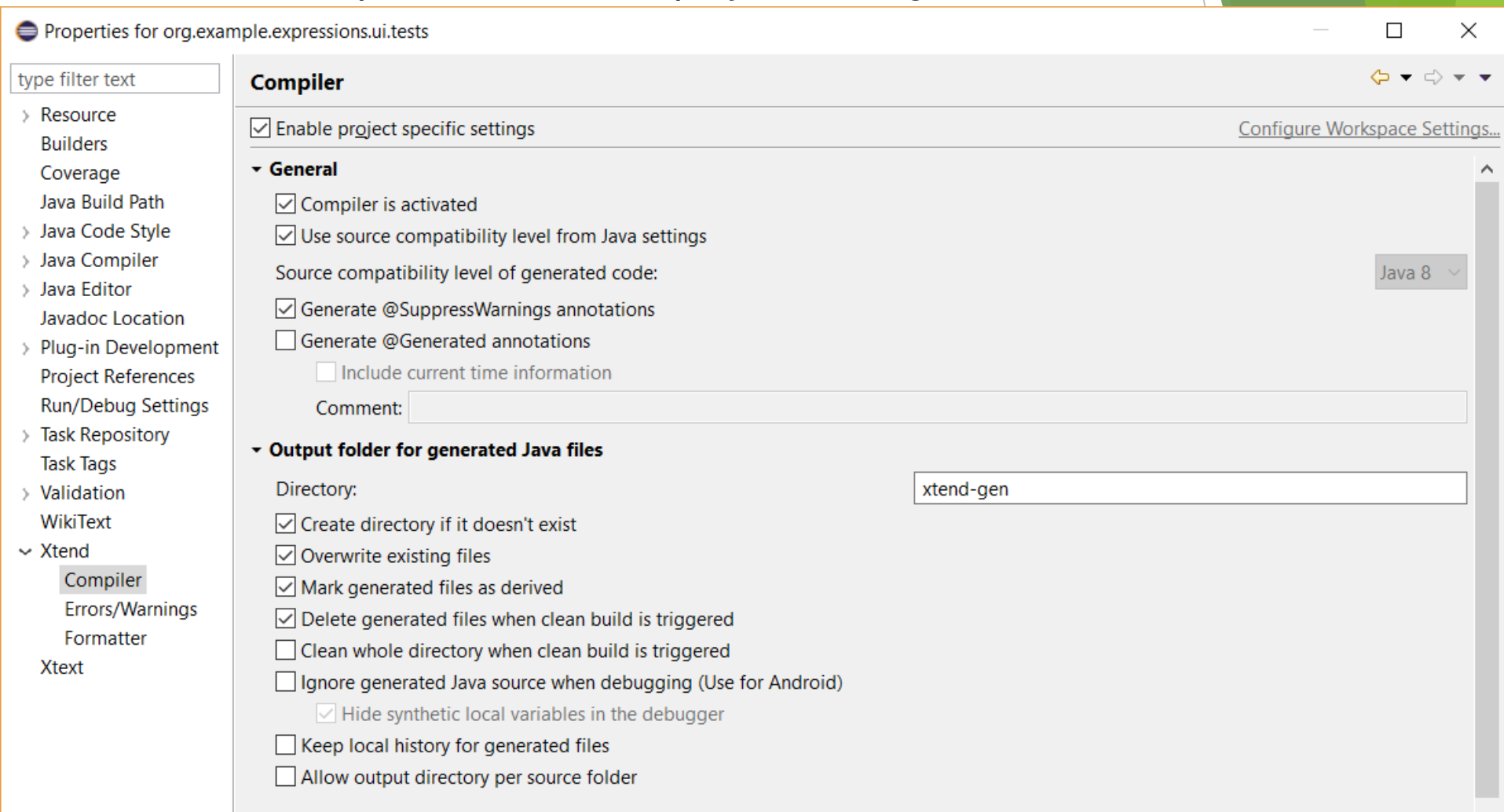
Infine realizziamo la **doGenerate**:

```
override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {  
    val program = resource.allContents.toIterable.filter(SJProgram).head  
    // an empty program is a valid SmallJava program  
    if (program == null)  
        return;  
  
    for (smallJavaClass : program.classes) {  
        fsa.generateFile  
            (smallJavaClass.fullyQualifiedName.toString.replace(".", "/") + ".java",  
  
            ...  
package smalljava.example  
public class «smallJavaClass.name» «IF smallJavaClass.superclass != null»  
    extends «smallJavaClass.superclass.compileTypeReference» «ENDIF» {  
    «FOR field : smallJavaClass.fields»  
    public «field.type.compileTypeReference» «field.name»;  
    «ENDFOR»  
    «FOR method : smallJavaClass.methods»  
    public «method.type.compileTypeReference» «method.name»  
        («method.params.map[compileParam].join(", ")») «compileBlock(method.body)»  
    «ENDFOR»  
    }  
    ...  
        )  
    }  
}
```

Non avendo implementato o lo scoping, sarà tutto public..

Generatore

Abilitiamo il compilatore Xtend dai project settings:



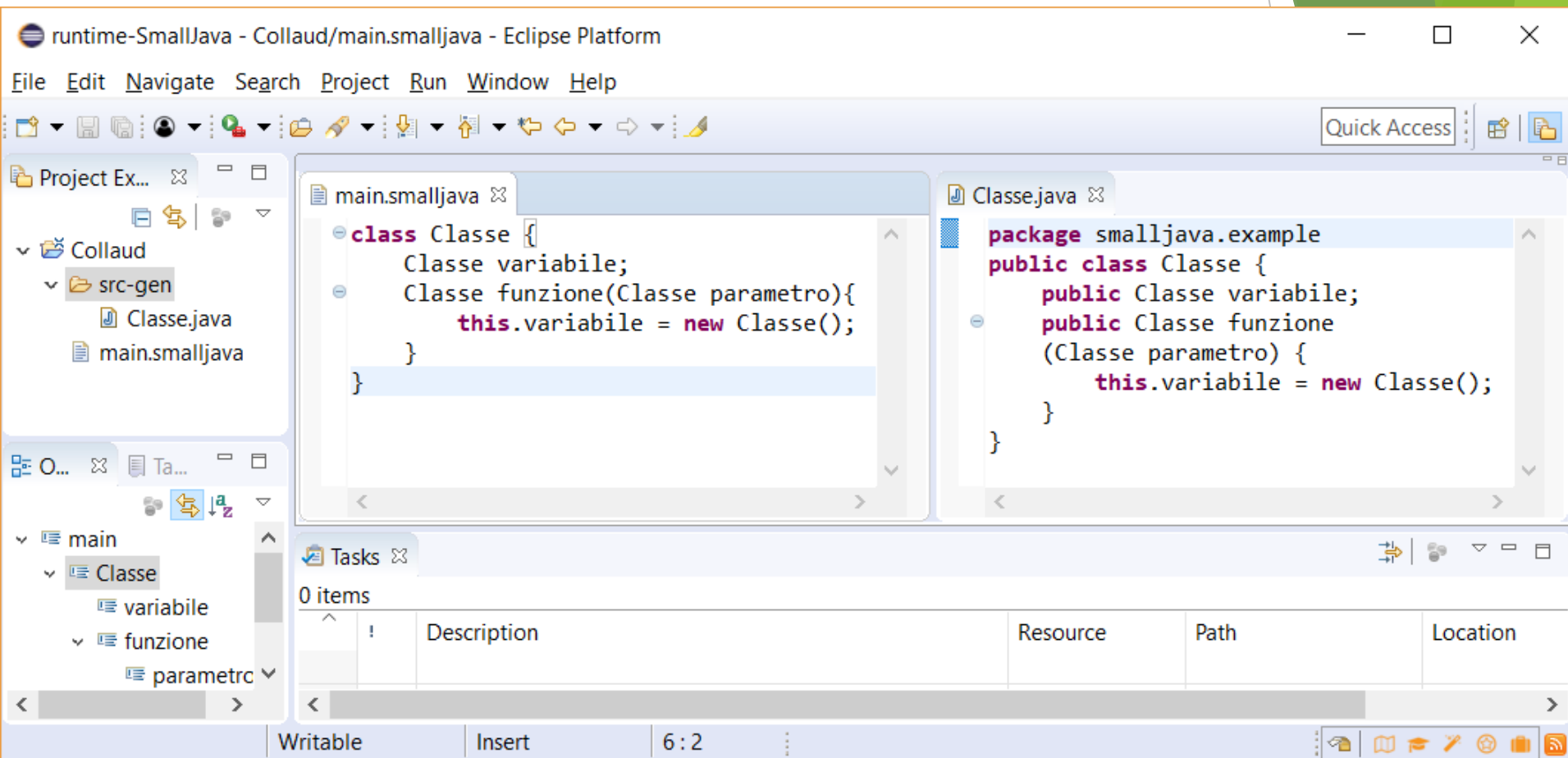
Generatore

Dopo aver ri-eseguito il MWE Workflow dovremmo avere un Main.xtend nel package del generatore. Dobbiamo modificarlo per eseguire la validazione del programma prima della generazione:

```
def protected runGenerator(String[] strings) {  
  // Load the resource  
  val set = resourceSetProvider.get  
  // Configure and start the generator  
  fileAccess.outputPath = 'src-gen/'  
  val context = new GeneratorContext => [  
    cancelIndicator = CancelIndicator.NullImpl  
  ]  
  
  // load the input files  
  strings.forEach[s|set.getResource(URI.createFileURI(s), true)]  
  // validate the resources  
  var ok = true  
  for (resource : set.resources) {  
    println("Compiling " + resource.URI + "...")  
    val issues = validator.validate(resource, CheckMode.ALL, CancelIndicator.NullImpl)  
    if (!issues.isEmpty()) {  
      for (issue : issues) {  
        System.err.println(issue)  
      }  
      ok = false  
    } else {  
      generator.generate(resource, fileAccess, context)  
    }  
  }  
  if (ok)  
    System.out.println('Programs well-typed.')  
}
```

Generatore

A questo punto dovrebbe essere tutto pronto per avere un generatore. Eseguendo Eclipse con SmallJava nella cartella src-gen verranno compilate le classi Java:



E adesso?

- Una cosa molto interessante sarebbe integrare Expressions in SmallJava, in maniera di avere un linguaggio ad oggetti che calcola le espressioni!
- Cosa più impegnativa, invece, è lo scoping, e quello richiede un ulteriore capitolo a parte..

Bibliografia

- Bettini, Lorenzo. *Implementing Domain Specific Languages with Xtext and Xtend*. Packt Publishing Limited, 2016.
- Gamma, E. Helm, R. Johnson, R. e Vlissides, J. Design patterns : elementi per il riuso di software a oggetti. Addison-Wesley : Pearson education Italia, 2002.
- Zarnekow, S. (2012). Xtext Best Practices. EclipseCon Europe, https://www.eclipsecon.org/2012/sites/eclipsecon.org.2012/files/Xtext_bestPractices.pdf