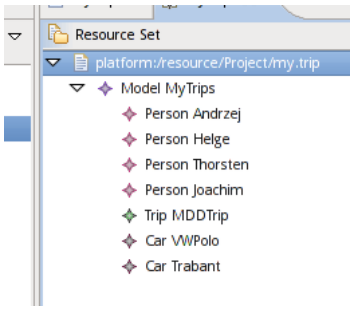
Ingredients

Reading: We will talk about parsing, grammars, left-recursion, etc. The Dragon book [1] is your friend, if you are rusty on these.

To get the idea of the Xtext framework you should read the first part (Getting Started) of the extensive documentation available at http://www.eclipse.org/Xtext/documentation/2.3.0/Documentation.pdf. In general familiarize yourself with http://www.eclipse.org/Xtext/, if your project involves implementing a textual DSL. Watch the videos, read the more advanced parts of the manual.

1 Introduction

A metamodel for a language can provide an abstract syntax. Domain experts and programs using this language will often need a readable concrete syntax, which can be manipulated in a text editor. Human readable syntax is sometimes also useful, even if models are not written by humans, but are created and processed completely automatically—human readable syntax is very helpful in debugging and monitoring in such cases.



You can think of the tree view editor, recalled in the figure above, as of an abstract syntax editor. While the diagram editor gets much closer to what a concrete syntax editor can do.

2 External DSLs with Xtext (Demo)

Xtext is the most popular language workbench for Eclipse. It is aims at languages with textual concrete syntax (as opposed to visual languages). We start with a simple demonstration: we will use Xtext to automatically derive concrete textual syntax for the abstract syntax of one of the examples used before in this course. The following EMF model describes the trip language, which can specify relations between trips, vehicles and passengers (persons). We used a similar example a few lectures ago:

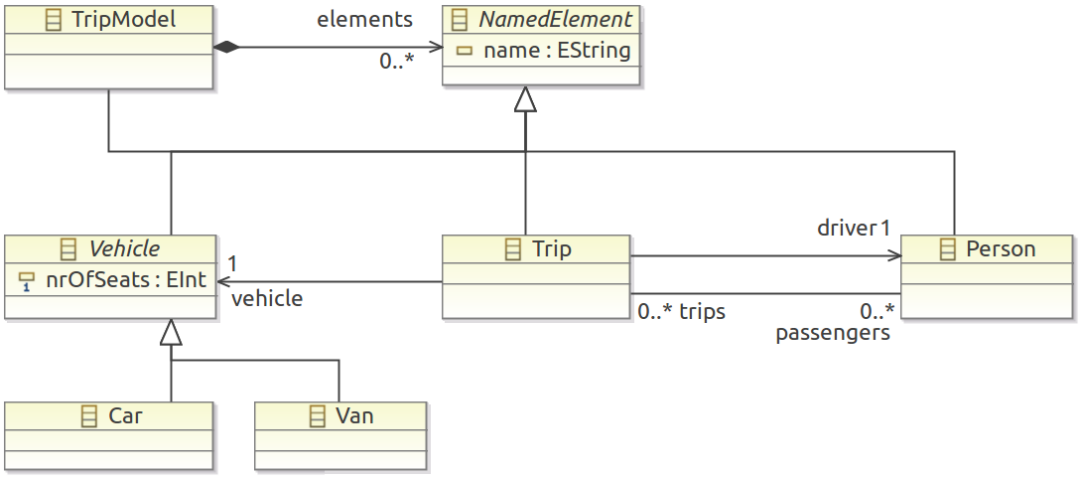


Figure 1: Abstract syntax (the meta-model) of the trip language

Before we proceed with generating a textual editor for this language, let me point out two properties of this model, which are necessary if you use Xtext. First, there is a root class, that represents the model (here this is TripModel). This class owns all the other elements through aggregation (containment). In this simple meta-model this ownership is direct, but it could also be indirect (via contained classes, that in turn contain other classes). Second, all important model elements are named. This often simplifies accessing them from various component frameworks. The name field is treated specially in the framework.

We follow these steps:

1. Create the .ecore file with the meta-model. Set the nsURI property of the toplevel package in the .ecore file to point to the location of the file in the Eclipse workspace (`platform:/resource/projectName/pathWithinProject/trip.ecore`)
2. Create the default generator model for the trip.ecore file (File / New / Other ... / EMF Generator Model).
3. Genereate the model code from this new genmodel (context menu of the TripModel in .genmodel editor)
4. Now create an Xtext project using the *Xtext Project From Existing Ecore Models* wizard (File / New / Project ...). Choose trip.genmodel generated above when asked for the

EPackage file in the wizard. Select TripModel as the root class (entry rule) in the same dialog.

5. A new project is created, and the default grammar specification for your language generated and opened in a text editor.
6. Generate Xtext artifacts (below). Approve downloading ANTLR.
7. Run the main xtext project as an Eclipse application.

The wizard generates four projects, out which the top one (without ui or test subpackage suffix) is of main interest. In the file trip.xtext you will find the syntax definition of our language (see Fig. 2).

Now, we generate the xtext artifacts (just a call from the *Run As...* context menu of the grammar editor). In a few seconds we can run the generated Eclipse application (plugin) and enjoy editing of trips with text highlighting, code completion, name resolution and interactive error reporting. See Figure 3.

The trip model in Fig. 3 is incorrect: Helge is not declared as a person. This is why the tool reports this immediately by underling the reference in the driver entry. The problem is also listed in the *Problems* view in the bottom of the screen (not shown in the figure). Technically this means that Xtext not only parses the model, but also performs static validation of references. It also does type checking, and can be integrated with other validity constraints. The underlined token is actually reference to an undeclared Person object.

Also note that terminals of the above grammar specification have been directly turned into keywords in the editor.

An attentive reader could notice that there is another (integrity) rule that is violated by the example model in the trip editor. Person blocks list a few trips, but Trip blocks do not list the respective persons in the passenger list. This violates the constraint that the passengers reference is the inverse (*EOpposite*) of the trips reference in our meta-model. Indeed, the editor does not check these constraints presently, although suitable EMF mechanisms could be invoked (programmatically).

Let us finish this section with a quote that Xtext project uses to introduce itself:

> *A quote from the guide: Xtext provides you with a set of domain-specific languages and modern APIs to describe the different aspects of your programming language. Based on that information it gives you a full implementation of that language running on the JVM. The compiler components of your language are independent of Eclipse or OSGi and can be used in any Java environment. They include such things as the parser, the type-safe abstract syntax tree (AST), the serializer and code formatter, the scoping framework and the linking, compiler checks and static analysis aka validation and last but not least a code generator or interpreter. These runtime components integrate with and are based on the Eclipse Modeling Framework (EMF), which effectively allows you to use Xtext together with other EMF frameworks like for instance the Graphical Modeling Project GMF.*

```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

import "platform:/resource/episode10xtext/model/trip.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

TripModel returns TripModel:
  {TripModel}
  'TripModel' name=EString
  '{'
    ('elements' '{' elements+=NamedElement ("," elements+=NamedElement)* '}')?
  '}';

NamedElement returns NamedElement:
  Trip | Person | Car | TripModel | Van;

Vehicle returns Vehicle:
  Car | Van;

EString returns ecore::EString:
  STRING | ID;

Trip returns Trip:
  'Trip' name=EString
  '{'
    'vehicle' vehicle=[Vehicle|EString]
    ('passengers' '('passengers+=[Person|EString]("," passengers+=[Person|EString])*')')?
    'driver' driver=[Person|EString]
  '}';

Person returns Person:
  {Person}
  'Person' name=EString
  '{'
    ('trips' '(' trips+=[Trip|EString] ( "," trips+=[Trip|EString])* ')' )?
  '}';

Car returns Car:
  'Car' name=EString
  '{'
    'nrOfSeats' nrOfSeats=EInt
  '}';

Van returns Van:
  'Van' name=EString
  '{'
    'nrOfSeats' nrOfSeats=EInt
  '}';

EInt returns ecore::EInt:
  '-'? INT;
```

Figure 2: Automatically derived concrete syntax definition, based on the trip metamodel.

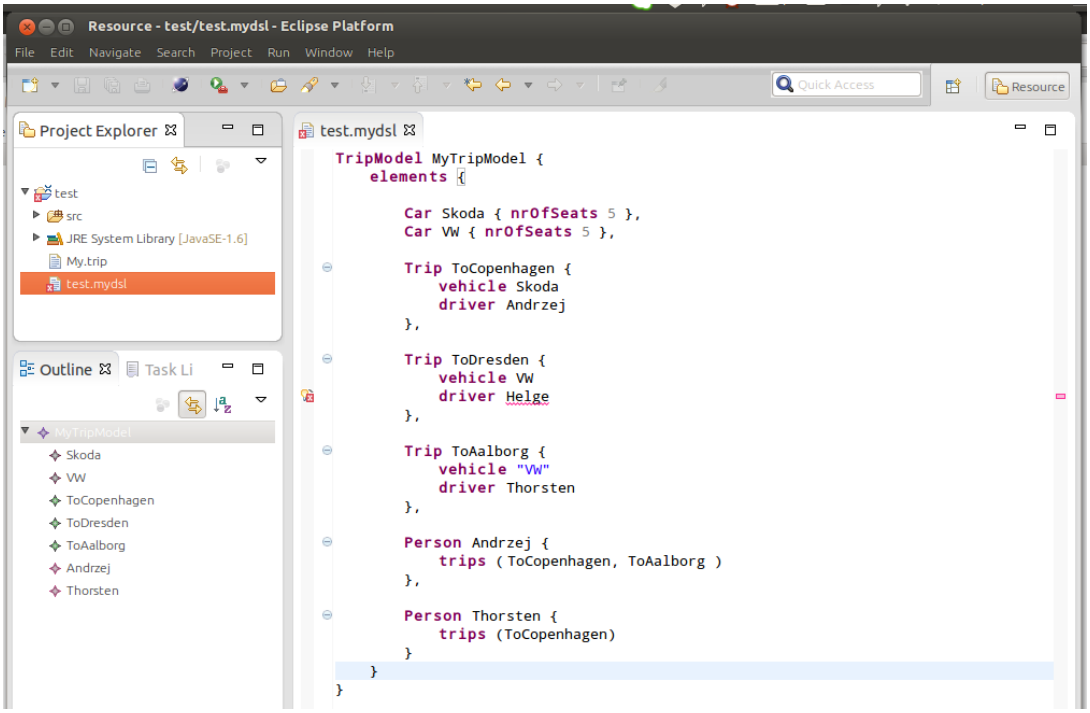


Figure 3: Generated default plugin for the trip language (in action)

> *In addition to this nice runtime architecture, you will get a full blown Eclipse-IDE specifically tailored for your language. It already provides great default functionality for all aspects and again comes with DSLs and APIs that allow to configure or change the most common things very easily.* (XText online documentation)

In the following section we will explain the syntax definition format of Xtext, so that you can define concrete syntax yourself. We will also rewrite the default grammar into a simpler, more human-friendly one. This will have the advantage that we will have two similar languages defined, which we will be able to use as an example for model transformations later in the lectures.

3 Concrete Syntax Definitions in Xtext

The XText specification language is a variation of the familiar *Extended Backus-Naur Form* (EBNF) notation for context free grammars, which is also used by most parser generators, so you remember it from your compiler course. Xtext relies on the ANTLR parser generator to produce parsers. ANTLR is a recursive descent parser so it can only handle LL(k) languages, where k is the size of lookahead. The parser is usually most efficient, if the k is small, preferably equal to one.

This restriction of ANTLR (and thus of Xtext) is not a serious one, as it is widely believed that all interesting programming languages are LL(k) languages, and can be parsed by such

parsers. The only problem is that it sometimes takes some effort to put the grammar of the language in the right form. Here the main problem is to make sure that the grammar is not *left-recursive*. Recall that a grammar is left-recursive if it has a production in which the left hand side nonterminal appears as the first element of the right hand side (either directly or indirectly via another nonterminal). Left-recursive grammars are quite common, for example a typical arithmetic expression grammar would include a production similar to

$$expr \rightarrow expr\ binary\text{-}operator\ expr\ ,$$

which is left-recursive. Sometimes a serious rewrite is necessary to remove left-recursion. For example:

$$\begin{gathered} expr \rightarrow term\ (+\ term)^* \\ term \rightarrow factor \mid factor\ *\ factor \\ factor \rightarrow ID \mid (\ expr\ ) \end{gathered}$$

The standard reference on parsing techniques, where you can also find left-recursion elimination is the so called *Dragon Book* by Aho and others [1], but almost any compiler construction text book would do.

We shall now discuss the main syntactic elements of the Xtext language. First the specification starts with the name of the grammar (in the same format as fully qualified Class name in Java):

```
grammar org.xtext.example.mydsl.MyDsl
```

In order to enable reuse in language definition, Xtext allows importing other grammars. In our example above, we have included the grammar that describes standard terminals in a typical programming language (the terminal `ID` used in Fig. 2 comes from this included grammar):

```
with org.eclipse.xtext.common.Terminals
```

Then we import the meta-models used as the abstract syntax:

```
import "platform:/resource/episode10xtext/model/trip.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

The first model imported is the model presented in Figure 1. We also import Ecore itself, in order to be able to use its types, for instance EString. Note that the elements of trip become available in the default name space (no *as* clause). So later in the grammar, types like Person are referred to directly. At the same time the Ecore types are prefixed by the name space `ecore`. Like in OCL, double colon is used as the name space prefix operator.

The first symbol (the left hand side of the first production) is the start symbol of the grammar. Here: `TripModel`. The `returns` construction allows to specify the type representing a nonterminal in the abstract syntax. The default type has the same name as the nonterminal, so most of the *returns* clauses in the generated grammar are redundant, but some are not.

Terminals are simply introduced as string literals. For example `'Car'` and the braces in the following rule:

```
Car: 'Car' name=EString '{' 'nrOfSeats' nrOfSeats=EInt '}';
```

A value resulting from parsing a nonterminal can be stored directly in a property of the current abstract syntax object. For example, the above production says that an object of type Car will be constructed upon its successful application. The name of the car (Car.name in Java) will be initialized with the value of a string directly following the first keyword. Similarly the number of seats will be initialized to an integer value following slightly later. In general a property of any time (including class types) can be assigned with an object constructed by invoked productions.

In the following rule we see how elements parsed can be used to populate collections.

```
Person returns Person:
  'Person' name=EString
  '{'
    ('trips' '(' trips+=[Trip|EString] ( "," trips+=[Trip|EString])* ')' )?
  '}';
```

If a property of an abstract syntax object is a collection, we can add a value to the collection (as opposed to replacing the collection) using the += operator instead of assignment. This happens for both vehicles and elements above. There is no null pointer error, since the collections are initialized to be empty upon object creation (by EMF).

The braced type name (`{TripModel}`) enforces creating an instance of a given type. This is useful if we are parsing a concept that is represented by an abstract type with several possible implementations. For example, parsing named elements, could be more concisely written without introducing nonterminals for Cars and Persons as:

```
NamedElement returns NamedElement:
      Trip
    | Car
    | TripModel
    | Van
    | {Person} 'Person' name=EString '{'
        ('trips' '(' trips+=[Trip|EString]
                  ( "," trips+=[Trip|EString])* ')' )?
'}';
```

The use of this construct in Fig. 2 is redundant (this is because this is automatically generated code, that needs to cater also for other complex meta-models).

Remaining syntax: alternatives (|), repetition (+,*), optionality (?) is familiar from most regular expression dialects, and the meaning is as expected.

A crucial ability of Xtext is resolving references.

The syntax for a cross-reference is `[TypeName|RuleCall]` where `RuleCall` defaults to `ID` if omitted. The parser only parses the name of the cross-referenced element using the ID rule and stores it internally. Later on, the linker establishes the cross-reference using the name, the defined cross-reference's type (Entity in this case) and the defined scoping rules. For example:

```
Person: 'Person' name=EString '{'
    ('trips' '(' trips+=[Trip|EString]
              ( "," trips+=[Trip|EString])* ')' )?
'}';
```

There is much more to Xtext, than I present, but this is sufficient to do simple languages. Among other elements, of the highest interest are probably customizable scoping semantics (what names are visible in what scopes), and fully qualified name support for references across name spaces/scopes. These are described in Xtext documentation.

To end this lecture we present another version of the grammar for the same language Trip that leads to a much simpler syntax:

```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

import "platform:/resource/episode10xtext.trip/model/trip.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

TripModel: (elements+=NamedElement)*;


NamedElement returns NamedElement:
    Trip | Person | Car | Van;

EString returns ecore::EString:
    STRING | ID;

Trip: 'trip' name=EString
        'car' vehicle=[Vehicle|EString]
        ('passengers' passengers+=[Person|EString] ( "," passengers+=[Person|EString])*)?
        'driver' driver=[Person|EString];

Person: 'person' name=EString;
Car: 'car' name=EString 'with' nrOfSeats=EInt ('seats' | 'seat');
Van: 'van' name=EString 'with' nrOfSeats=EInt ('seats' | 'seat');

EInt returns ecore::EInt: '-'? INT;
```

It took me not more than 10 minutes to rewrite the default generated syntax specification to this one, including generating a new Xtext based editor for models. An example model in this syntax looks as follows:

```
person Andrzej
person Helge
person Thorsten
person Joachim

car VWPolo with 4 seats
car Trabant with 4 seats

trip MDDTrip
    car VWPolo
    passengers  Helge, Andrzej, Thorsten, Joachim
    driver Joachim
```

In fact this model violates the meta-model constraint, because with this parser it only stores the references from trips to passengers, and not the other way around (remember that we have an integrity constraint that makes the two references inverse of each other).

So loading this model into EMF infrastructure (depending on configuration) may cause validation errors. The idea is that the inverse reference should not be stored in the model, but can be recovered with a model transformation. We will use this as an example in later classes.

4 DSL Design Guidelines (Concrete Syntax)

It is known that 75% of population prefers visual to textual syntax. Here Xtext is not very useful [4]. However state of the art in applications of DSL is using them by engineers to write models *together* with subject matter experts. Here textual DSLs are good enough. Use of DSLs by subject matter experts alone is rare.

It is also known that programmers are more likely to prefer textual notations (to visual), so DSLs aimed at software developers should probably be textual. [4]

Choosing either representation purely on the basis of prejudice is bad, as is ignoring other possibilities such as matrices, tables, forms, or trees. The correct representational paradigm depends on the audience, the data's structure, and how users will work with the data. Making the wrong choice can significantly increase the cost of creating, reading, and maintaining the models. [4].

Whenever possible you should adopt existing notations used by the intended audience of your language. It is known that we engineers are fast in (and attracted to) learning new languages. This is completely opposite with non-programmers. Introducing new notation is likely just another barrier to acceptance of technology you want to introduce (on top of the tool and process change) [3]. For the same reason avoid changing the meaning of known symbols (for example + should mean addition, and unlikely anything else). If in need for new symbols, use descriptive terms (English words).

Always allow comments. These are very useful for evolving and experimenting with models [3].

It is very practical to keep the abstract and concrete syntax not too far apart. This simplifies your implementation (parsing, transformations, pretty printing). Element with different concrete syntax should have different abstract syntax. Also the concrete syntax should be context independent—concrete syntax of an element should be dependent on its abstract type, and not on the context of use [3].

Karsai and coauthors [3] give many other very practical guidance, thus I again encourage you to read their paper when designing your DSLs.

References

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. *Compilers: Principles, Techniques, and Tools. Edition 2.* Prentice Hall, 2006.

[2] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages.* Addison-Wesley, 2011.

[3] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design guidelines for domain specific languages. In *9th OOPSLA Workshop on Domain-Specific Modeling*, 2009.

[4] Steven Kelly and Risto Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29, 2009.