

Tutorial for using ANTLR for MPS project -- See this project at
https://github.com/campagnelaboratory/ANTLR_MPS

This tutorial aims to get you started with ANTLR for MPS. Before you follow this tutorial, you will need to obtain:

The ANTLR grammars-v4 repo, which contains the **CSV** file format grammar we are going to use in this example. It can also be convenient to use the IntelliJ ANTLR plugin to generate the Lexer and Parsers, as Java source code from the grammar. Let's have a look at the CSV grammar, a simple grammar to describe the content of comma separated value file formats:

grammar CSV;

file: hdr row+ ;

hdr : row ;

row : field (',' field)* '\r'? '\n' ;

field

: TEXT

| STRING

|

;

TEXT : ~[\n\r"]+ ;

STRING : '"' (""""|~'')* '"' ; *// quote-quote is an escaped quote*

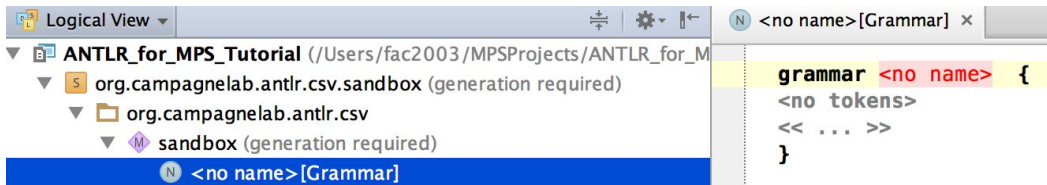
The grammar defines four parser rules: file, hdr, row and field. Note that parser rule names are written in lower-case. The grammar also defines two lexer rules: TEXT and STRING, whose names are written in upper-case, by convention.

Step 1. Materialize the grammar into MPS

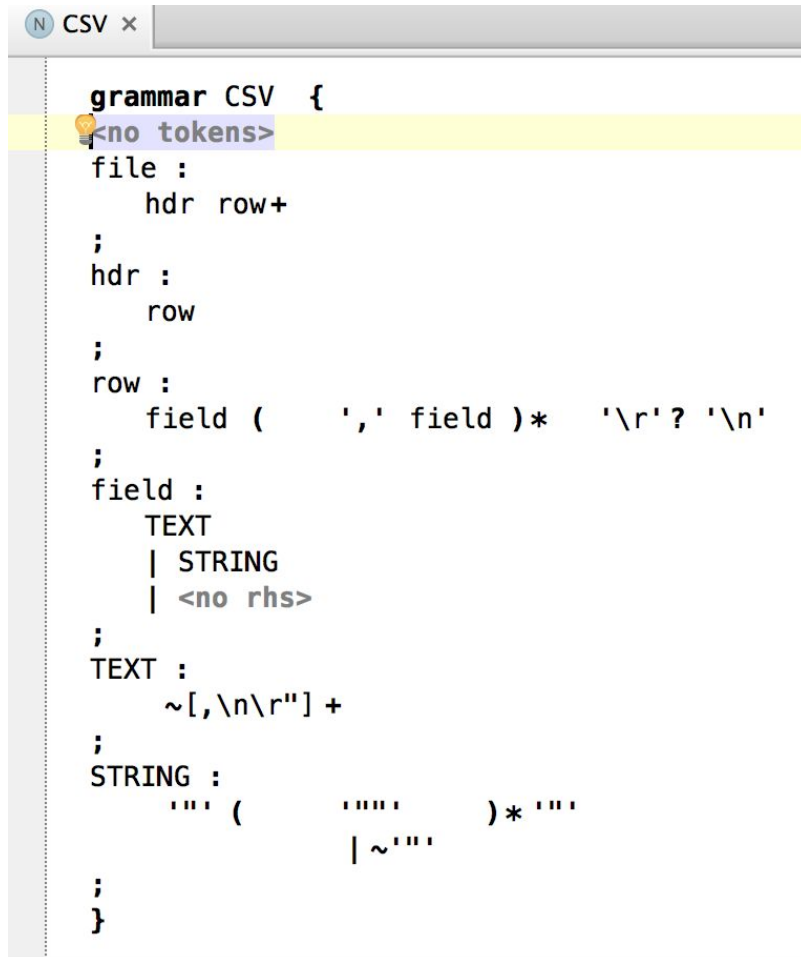
As a first step, we will materialize the grammar into MPS. This will make it possible to view the rules in MPS and to refer to them from other languages and solutions. To this end, create an MPS solution and import the language *org.campagnelab.ANTLR* into a model of this solution:



Right-click on the sandbox model and create a new Grammar AST Root node:



To materialize the grammar in MPS, place the cursor over the mps place holder << ... >>, right click and select Paste ANTRL Rules, with the text of the CVS grammar in the clipboard. This should result in:



The rules are laid-out a bit differently than in the text version of the grammar, but now can be connected to other languages and manipulated with the projectional editor (e.g., intentions can be added to rules or alternatives).

Let's try an intention. Select the field rule, and use 'Label All Alternatives'. This intention will add a label to the alternatives of the Field rule. Each alternative is named after the text of the lexer rule reference. The last label is left empty because nothing could be used to determine the label. Let's call this one EMPTY_FIELD. After naming this label, the field rule should look like this:

```
field :
  TEXT # TEXT
  | STRING # STRING
  | <no rhs> # EMPTY_FIELD
;
```

Defining labels make it possible to differentiate each alternative of the field Rule and this is useful when creating a visitor that handles alternatives differently.

Note that this change must be transferred back to the textual grammar. Select the field rule and copy it to the clipboard. You can then paste its text to IDEA to replace the field rule in the text version of the grammar (in this case, remove the “<no rhs>” tip to keep the grammar valid for ANTLR). After transferring a change back to text, remember to regenerate the lexer and parsers. The parser must be in sync with the MPS version of the grammar.

Step 2. Define a Grammar to MPS Converter

In the second step, we will define the mapping from the grammar and the parse tree produced by the parser to an MPS language. To this end, you need to import the *org.campagnelab antlr.tomps* language into the sandbox model. Once you have done this, create an AST root node of type *o.c.antlr.tomps.ConvertToMPS*.

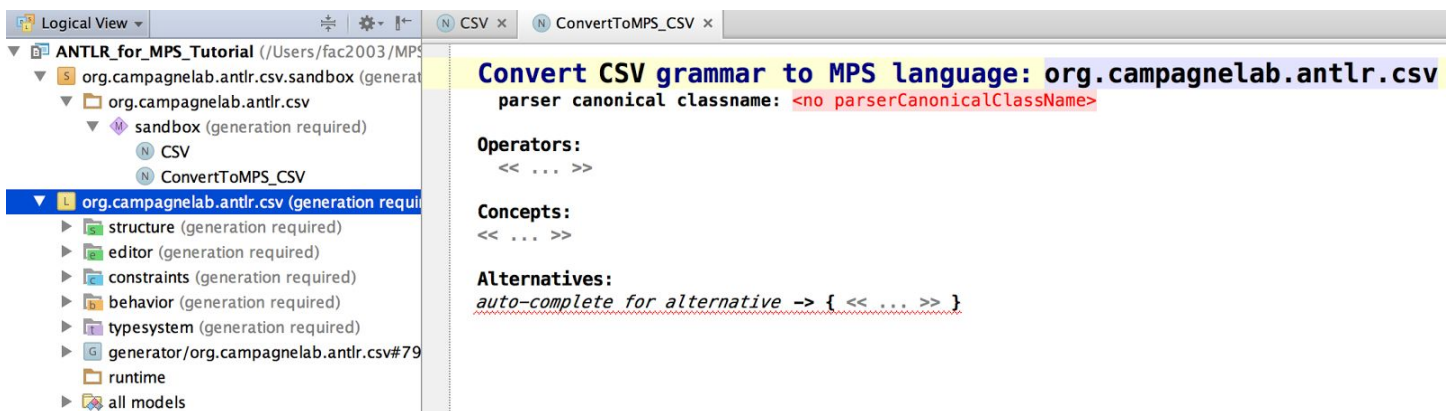
The new AST root node will appear as shown:



You can bind the converter to the grammar. Place the cursor over <no grammar> and locate the CSV node.

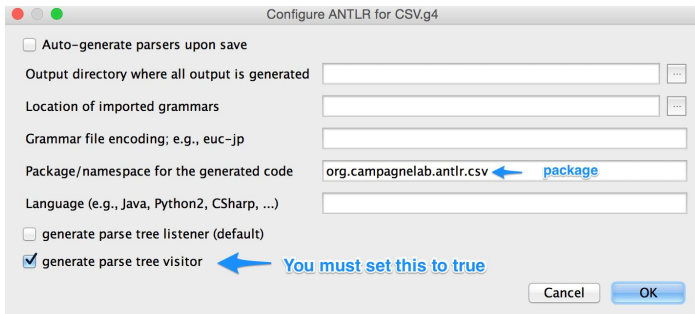
Step 3. Define a target language.

In this step, we create a language to hold the nodes/information that will be converted from the grammar. Let's call this language *org.campagnelab.antlr.csv*. Create this new language in the project (leave the language alone for now, there is no need to customize it in this step) and type its name in the converter (top-right). The project will then appear as shown here:



Step 4. Set the name of the parser class.

Configure ANTRL to put the lexer and parser implementations in a package and to generate interfaces for visitors. After you have done this (hint, the IDEA ANTLR plugin is convenient), assemble a JAR file with the lexer and parser classes.



Given this configuration, the parser class will be called `org.campagnelab.antlr.csv.CSVParser`. Type this into the converter (parser canonical class name attribute).

Now would be a good time to create the jar file that contains the lexer/parser implementation and add it to the destination language (under a model root of type `java_classes`). Adding the jar to the language will ensure that you can preview or generate the visitor implementation and resolve references to the parser.

Step 5. Map Rules to Concepts.

Place the cursor over the mps place-holder under Concepts: and press-return to create a concept mapper. Use auto completion to locate the file parser rule. The rule is visible because you have connected the converter to the CSV grammar. After you have linked the mapper to the rule, the Concepts attribute will appear as:

Concepts:

```
file -> concept< <no concept> >
```

There is an intention attached to the concept mapper that you can use to create the concept in the target language. The intention is called ``Create Concept(s)``. You can use it with one or more mappers. Any concept that is not yet assigned will be created in the target language. The Concepts view changes to:

Concepts:

```
file -> concept< File >
```

The name of the concept appears in red because the model is missing a dependency on the structure aspect of the destination language. Open the model dependency for sandbox and add `org.campagnelab.antlr.csv.structure`. Doing this will resolve the dependency and the File concept will be shown in black and hyperlink to the concept in the destination language (command/ctrl+B).

Continue to create mappers, one for each parser rule of the grammar, and use the ``Create Concept(s)`` intention until you have the following:

Concepts:

```
file -> concept< File >
hdr -> concept< Hdr >
row -> concept< Row >
field -> concept< Field >
```

Step 6. Map alternatives

Some rules provide alternatives, shown with the `|` character in the rule. You can create specific mappers for each alternatives using the intention ``Add All Alternatives for Concept X``, where X represents the name of the

concept. Try it by defining alternative mappers for the concept Field (place the cursor over field under Concept:, or over the name of the concept, Field). You should then see the following:

Alternatives:

```
field: TEXT -> { << ... >> }  
field: | STRING -> { << ... >> }  
field: | -> { << ... >> }
```

Note that you can define sub-concepts corresponding to each alternative using the “**Create Concept(s) and Labels**” intention. To demonstrate this, let’s start with the field rule (you must have mapped alternatives first, which we just did for field). Place the cursor over field under Concepts: and invoke the “**Create Concept(s) and Labels**” intention. This will associate each alternative to a concept

Alternatives:

```
field: TEXT -> concept<TextField> { << ... >> }  
field: | STRING -> concept<StringField> { << ... >> }  
field: | -> concept<EmptyField> { << ... >> }
```

Note that the sub-concepts are named after the labels defined in the grammar. For this reason, it is a good idea to use CamelCase labels in the grammar. For instance, Use Text, rather than TEXT when defining the label for the TEXT alternative. Note that this would be done automatically for parser rules that already support this convention, but must be adjusted manually if for lexer rules (e.g., TEXT and STRING shown in this example).

Notice that you can rename each concept as you would normally do in MPS and the reference in the converter to the concept will be automatically be updated. For instance, I chose to rename the concepts to follow CamelCase:

Alternatives:

```
field: TEXT -> concept<TextField> { << ... >> }  
field: | STRING -> concept<StringField> { << ... >> }  
field: | -> concept<EmptyField> { << ... >> }
```

Step 7. Define Properties and Children

At this time we can add properties to the Field sub-concepts which are necessary to store information exposed in the ANTLR ParseTree (this data structure will be created automatically by the parser generated with ANTLR). The ANTLR.tomps language does not automate this step because you should spend time thinking about how you want the MPS AST to be organized. For instance, for the Field concept tree, there are several choices for storing the value that will be read. You could have one property for each sub-concept TextField and StringField, or a single property in the Field concept, which both sub-concepts see. Note that the only difference between StringField and TextField is that StringField values are enclosed in double-quotes (see CSV grammar). It thus makes sense to store the value the common parent concept Field. Define a value property as shown here:

```
concept Field extends <default>
      implements <none>
```

```
instance can be root: false
alias: <no alias>
short description: <no short description>
```

```
properties:
value : string
```

In this step, you need to design the MPS structure of the destination language. The support that the *tomps* language provides is to help you keep track of which concepts map to which rules, and to create sub-concepts for rule alternatives (setting up the concept alias using keywords found in the grammar, when appropriate). You are responsible for designing the AST structure that will best fit the needs of the language user.

Step 8. Define mapping operations for each rule alternative

In this step, we define alternative mappers for each rule alternative. Let's continue with the alternatives of the Field concept. The alternative currently has no mappers defined:

Alternatives:

```
field: TEXT -> concept<TextField> { |< ... > }
```

Put the cursor over the mps place holder and press return. This will add a mapper slot, as shown below:

Alternatives:

```
field: TEXT -> concept<TextField> {
  | -> |
}
```

Each mapper slot has a left part, a -> to denote the direction of the mapping, and a right-part. The left-part will auto-complete to elements on the alternative. In this example, you will be able to choose the TEXT field because it is part of the field: TEXT alternative.

The right side of the mapper will auto-complete to structure elements of the TextField concept. In this case, because we have defined a value property in the super-concept Field of TextField, you will see value as a property where the TEXT value can be stored. Select value on the right hand-side. The mapper will then look like this:

Alternatives:

```
field: TEXT -> concept<TextField> {
  TEXT:0..1 -> value
}
```

Notice the cardinality indicator on the right. You may use this to indicate that TEXT is a required element (change the cardinality from 0..1 to 1). Changing the cardinality to 0..n is useful to transfer values that can occur more than once in the grammar. The *tomps* language will generate the appropriate transfer code using the multiplicity that you define in the converter. Setting appropriate multiplicity is important because ANTLR generates a different API on the parser for elements of the grammar that can occur at most 1 or that can occur multiple times.

Remark: Note that you can preview the implementation that will be generated for the ANTLR visitor using Preview Generated Text in the MPS context-menu (select a node of the converter, right-click and choose Preview Generated Text).

Proceed to define a mapper for the StringField concept. The Alternatives section should then read:

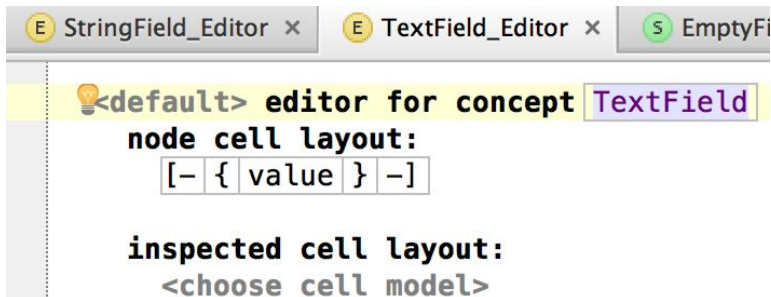
Alternatives:

```
field: TEXT -> concept<TextField> {  
  TEXT:0..1 -> value  
}  
field: | STRING -> concept<StringField> {  
  STRING:0..1 -> value  
}  
field: | -> concept<EmptyField> { << ... >> }
```

Note that the right-hand side of the mapper provide intentions useful when you need to convert the value on the left to the type of the value on the right. For instance, you can use the intention called “Convert to boolean” to convert a string to a boolean value to be stored in a property of type boolean. Such transformations can be chained, so in the future it would be possible to define an intention that calls a behavior method on the destination concept to convert some value in some specific way.

Step 9. Creating Editors

Let’s now create editors for the Field sub-concepts. Place the cursor over the Field concept in the Concepts: section, and use the intention “Create Editors”. This will produce editor implementations into the editor aspect of the target language. For instance, the TextField editor will look like this:



The screenshot shows a code editor with three tabs: "StringField_Editor", "TextField_Editor", and "EmptyFi". The "TextField_Editor" tab is active. The code in the editor is as follows:

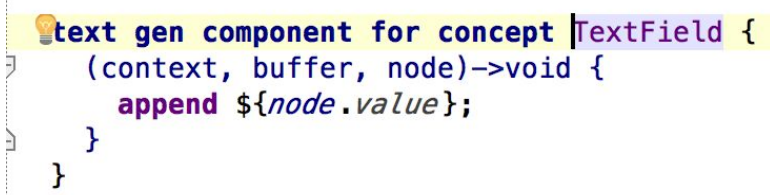
```
<default> editor for concept TextField  
node cell layout:  
  [- { value } -]  
  
inspected cell layout:  
  <choose cell model>
```

The editor is configured to display the value property because this property is mapped for the alternative corresponding to TextField. If there were several properties to display, the order of the data defined by the grammar would be followed to arrange the property/children in the editor. Note that you should manually inspect each generated editor and/or test it with real data to make sure the view is appropriate for the target language.

Step 10. Creating TextGen

Similarly to editor, tomps can help generate the TextGen aspect of the language. However, **before you proceed, you must create a TextGen aspect in the destination language**. If you fail to do so, the following will appear to do nothing.

For instance, place the cursor on top of the Field concept and use the intention “Create TextGens”. This will yield:



The screenshot shows a code editor with the following code:

```
text gen component for concept TextField {  
  (context, buffer, node)->void {  
    append ${node.value};  
  }  
}
```

Make sure you review all TextGen components because the mapping will often need attention (e.g., choices in the grammar will not be handled and both alternatives will be written).

Step 11. Using the Visitor in Client Code

When you have configured the converter, a visitor will be generated directly. Preview the generated code and will find that the visitor extends `CSVBaseVisitor<SNode>`. This class exists in the `org.campagnelab.antlr.csv@java_stubs` model, which was created when you imported stubs for the Jar file defined earlier (see Step 4).

Assume that you need to produce a class with a main method that will use the visitor (this is unlikely because such a class would not have access to the MPS environment, but will make this tutorial more concise).

```
public class MainClass {

    public static void main(string[] args) {
// obtain the parseTree using the ANTRL parser. The technique may vary based on
// the specific grammar you work with.

ANTLRInputStream input = new ANTLRInputStream(new FileReader(args[0]));
CSVLexer lexer = new CSVLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
CSVParser parser = new CSVParser(tokens);
CSVParser.FileContext treeContext = parser.file();

// then use the visitor to convert the parse tree to node of type File
    CSVBaseVisitor visitor = new visitor ConvertToMPS_CSV;
    node<File> file=(node<File>)visitor.visitFile(parseTree);
    }
}
```

The expression `new visitor` has a slot that you can use to locate the converter that will generate the visitor. Once configured, this expression will generate to an expression that creates an instance of the generated visitor.