emftext

USER GUIDE

August 10, 2010

Contents

1 Overview

EMFText is a tool which allows to define a textual syntax for Ecore based metamodels. From a specification of this syntax, EMFText generates components to load, edit and store model instances. The syntax is specified by a so called concrete syntax specification which are stored in files with the suffix `.cs`. A `cs` specification is directly related to one or more Ecore metamodel(s) which implicitly predefine a "grammar skeleton". Figure 1.1 gives an overview on how the generator part of EMFText works and which components are generated.

Through combining metamodel and `cs` specification, EMFText derives a context-free grammar and exports it as an Another Tool for Language Recognition (ANTLR) [**?**] parser specification. This specification contains annotated semantic actions which perform the instantiation of models. EMFText then transparently delegates parser[1] and lexer[2] generation to ANTLR by passing the generated grammar file. Since ANTLR does not cover the whole class of context-free grammars one can not guarantee the generation of a working parser for arbitrary cases. However, in most cases generation is sufficient.

While parsers are used to load model instances from textual representations a printer is needed to do the inverse (e.g., to print an in-memory model) back to a textual representation. The printed results can then again be parsed by the corresponding parser. Both instances (printed and loaded) are be equal. Furthermore, printers produce a formatted and human-readable output. The EMFText built-in printer generator achieves these goals by interpreting the `cs` file and the derived grammar. Also, `cs` specifications can be enriched by special operators to indicate that on a specific position white-spaces or newlines have to be printed. Additionally, it uses information about literals (e.g., keywords) in defined languages which are removed from model instances. As for parser generation, EMFText does not guarantee that printer generation works for arbitrary cases but mostly it is be a convenient solution.

EMFText also generates a set of resolvers. Resolvers convert parsed token strings to an adequate representation in the metamodel instance.

[1] http://en.wikipedia.org/wiki/Parser
[2] http://en.wikipedia.org/wiki/Lexical_analysis

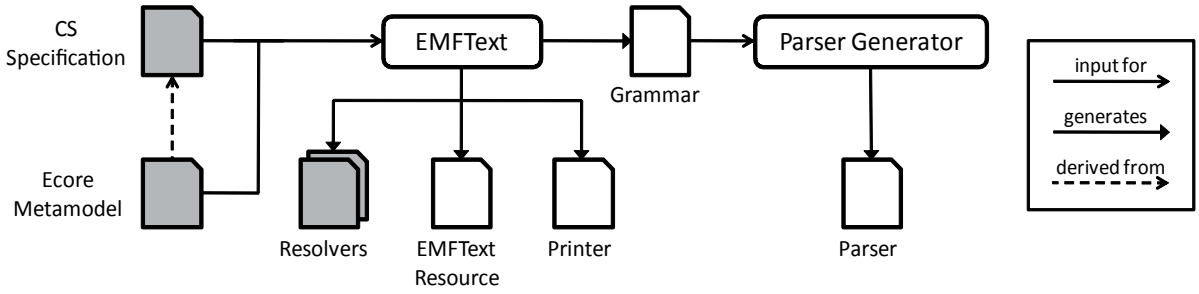

Figure 1.1: Overview of Specified and Generated Artefacts

TokenResolvers implement a mapping from the string value of a specific token to a native Java type (e.g., boolean, int, String etc.). In the standard implementation TokenResolvers can automatically remove and add (printing) pre- and suffixes. The conversion to native java data types is done by delegation to the corresponding Java type conversion functions. For example, `Integer.parseInt("42")` results in an int valued `42`. Since this behavior is only desired for concrete syntaxes mirroring exactly (or at least partially) the Java syntax for primitive types, users are expected to implement more adequate mapping functions as needed.

Resolvements depending on context are meant to be realised by implementing ReferenceResolvers. For these only stubs are generated. While TokenResolvers are directly invoked by parsers, ReferenceResolvers are triggered on demand later by EMF's proxy resolution mechanism. An additional feature is the evaluation of eventually annotated Object Constraint Language (OCL) [**?**] constraints. With OCL, consistency conditions can be declared on the metamodel to further improve quality of EMFText based developments.

2 Development Process

Creating parsers, printers and editors with EMFText is easy! It involves some necessary steps which are:

- creating a metamodel,
- specifying concrete syntax,
- generating resource plug-ins,
- customizing the DSL tooling.

Each of this steps will be explained in the subsequent sections.

2.1 Creating a Metamodel

The starting point usually is the *Ecore model*. It serves as abstract syntax and as a skeleton for concrete syntax. The model can be a graph or tree definition. Although Ecore models are specified as XML files, it is recommended to use the Ecore model editor or a graphical editor (e.g., the Ecore Tools). Additionally, a unique namespace (property NS URI in the .ecore file) for the models as well as a package name needs to be specified. To enable EMFText to use models at runtime, a model plug-in must be generated. For this, EMF provides a *generation metamodel* (GenModel) allowing to enrich Ecore models with specific information for code generation. The EMF Model wizard facilitates the creation of GenModels.

The next step is to define the base package name which will be the common prefix for all subsequently generated Java packages. After finishing configuration, the *EMF model plug-in* is ready to be generated. EMFText will later use these classes to construct a model instance.

2.2 Specifying Concrete Syntax

After defining a metamodel, we can start specifying our *concrete syntax*. As a starting point, EMFText provides a syntax generator that can automatically create a concrete-syntax specification using HUTN (Human-Useable Textual Notation) from the metamodel. To manually specify the concrete syntax, a textual specification has to be written, which has the ending `cs` (from **c**oncrete **s**yntax).

It consists of four sections: In the first part, a unique name refers to the syntax as a specific resource type. Furthermore, the metamodel, whose textual syntax shall be defined, has to be referenced by its unique namespace URI and a start symbol must be selected from the model elements. In the second part, productions from other concrete syntax specifications can be imported. This is especially useful if the metamodel is compositionally structured, e.g. reuses large parts from other models which have an already specified concrete syntax. The third part allows for specifying token types as it is usual for parser generators, but in contrast, EMFText also allows to leave them out: If no token definitions are given, default rules are used. Another

specialty for token definitions are optional pre- and suffixes which are transparently removed (after parsing) and added (before printing) by the generated token resolvers. And last but not least, EBNF-like productions have to be defined for each non-abstract model element reachable from the start symbol. They have to be defined with respect to the attributes and references of the model element and thus do not contain nonterminals in the classic sense.

2.3 Generating Resource Plug-ins

The context menu on concrete syntax specification files offers an item to generate the actual *resource plug-in* which contains the parser, printer and the editor for the language. In an optional last step, the generated token resolvers and the printer can be tailored to specific needs.

2.4 Customizing the DSL Tooling

TODO Add content

3 Concrete Syntax Specification Language (CS)

An EMFText syntax specification must be contained in a file with the extension `.cs` and consist of four main blocks:

1. A mandatory configuration block, which specifies the name of the syntax (i.e., the file extension), the generator model where to find the metaclasses, and the root metaclass (start symbol). Optionally other syntaxes and metamodels can be imported and code generation options can be specified.
2. An (optional) `TOKENS` section. Here, tokens for the lexical analyser can be specified.
3. An (optional) `TOKENSTYLES` section. Here, the default style (i.e., color and font style) for tokens and keywords can be specified.
4. A `RULES` section, which defines the syntax for concrete metaclasses.

In the following sections, these four main blocks will be explained in more detail.

3.1 Configuration Block

3.1.1 Required General Information

The first required piece of information, is the file extension that shall be used for the files, which will contain your models:

```
SYNTAXDEF yourFileExtension
```

Note: The file extension must not contain the dot character.

Second, EMFText needs to know the EMF generator model (`.genmodel`) that contains the metaclasses for which the syntax will be used. EMFText does use the generator model rather than the Ecore model, because it requires information about the code generated from the Ecore model (e.g., the fully qualified names of the classes generated by EMF). The genmodel is found using its namespace URI:

```
FOR <yourGenModelNamespaceURI>
```

To find the generator model with the given namespace URI, EMFText uses the EMF generator model registry. Also, EMFText looks for a `.genmodel` file, which has the same name as the syntax definition. For example, if the syntax specification is contained in a file `yourdsl.cs`, EMFText looks for a file called `yourdsl.genmodel` in the same folder.

Optionally, a second parameter (`yourGenmodelLocation`) can be given:

```
FOR <yourGenModelNamespaceURI> <yourGenmodelLocation>
```

This (second) parameter is only required, if the generator model is not registered in the generator model registry, or if the name of the syntax definition (without the file extension) is

different from the generator model file name. In both cases a relative or absolute URI can be given to point to the generator model.

Third, the root element (start symbol) must be given. The root element must be a metaclass from the metamodel:

```
START YourRootMetaClassName
```

A CS specification can also have multiple root elements, which must be separated by a comma:

```
START RootMetaClass1, RootMetaClass2, RootMetaClass3
```

Altogether a typical header for a `.cs` file looks something like:

```
SYNTAXDEF yourFileExtension
FOR <yourGenModelNamespaceURI> <yourGenmodelLocation>
START YourRootMetaClassName
```

3.1.2 Importing other Metamodels and Syntax Specifications

Metamodels and syntax specifications can be imported in a dedicated import section, which must follow after the start symbols:

```
IMPORTS {
    // imports go here
}
```

The list of imports must contain at least one entry. If no imports are needed the whole section must be left out. An import entry consists of a prefix, which can be used to refer to imported elements in rules, the metamodel namespace URI and optionally the name of a concrete syntax defined for that metamodel. If a syntax is imported, all its rules are reused and need not to be specified in the current `cs` specification. Importing syntax rules is optional. One can also just import the metamodel contained in the generator model.

```
prefix : <genModelURI> <locationOfTheGenmodel>
    // next line is option (except the semicolon)
    WITH SYNTAX syntaxURI <locationOfTheSyntax>;
```

The two locations are again optional. For resolving the generator model the same rules as for the "main" generator model (declared after the `FOR` keyword) apply. For locating the syntax, EMFText looks up the registry of registered syntax specifications. If no registered syntax is found, `locationOfTheSyntax` is used to find the `.cs` file to import. Again, `locationOfTheSyntax` must be a relative or absolute URI.

3.1.3 Code Generation Options

EMFText's code generation can be configured using various options. These are specified in a dedicated optional `OPTIONS` section:

```
OPTIONS {
    // options go here in the following form:
    optionName = "optionValue";
}
```

The list of valid options and their documentation can be found in Appendix A.

3.2 Tokens

EMFText allows to specify custom tokens. Each token type has a name and is defined by a regular expression. This expression is used to convert characters from the DSL files to form groups (i.e., tokens). Tokens are the smallest unit processed by the generated parser. By default, EMFText implicitly uses a set of predefined standard tokens, namingly:

- `TEXT : ('A'..'Z'|'a'..'z'|'0'..'9'|'_'|'-')+`,
- `LINEBREAK : ('\r\n'|'\r'|'\n')`,
- `WHITESPACE : (' '|'\t'|'\f')`.

The predefined tokens can be removed using the `usePredefinedTokens` option:

```
OPTIONS {
    usePredefinedTokens = "false";
}
```

3.2.1 Defining Custom Tokens

To define custom tokens, a `TOKENS` section must be added to the `.cs` file. This section has the following form:

```
TOKENS {
    // token definitions go here in the form:
    DEFINE YOUR_TOKEN_NAME $yourRegularExpression$;
}
```

Every token name has to start with a capital letter. A regular expression must conform to the ANTLRv3 syntax for regular expressions (without semantic annotations). However, don't worry: EMFText will complain if there is a problem with your regular expressions.

3.2.2 Composed Tokens

Sometimes, regular expressions are quite repetitive and one wants to reuse simple expressions to compose them to more complex ones. To do so, one can refer to other token definition by their name. For example:

```
TOKENS {
    // simple token
    DEFINE CHAR $('a'..'z'|'A'..'Z')$;
    // simple token
```

```
    DEFINE DIGIT $('0'..'9')$;
    // composed token
    DEFINE IDENTIFIER CHAR + $($ + CHAR + $|$ + DIGIT + $)*$;
}
```

If token definitions are merely used as "helper" tokens, they can be tagged as FRAGMENT. This means the helper token itself is used in other token definitions, but not anywhere else in the syntax specification:

```
TOKENS {
    // simple token
    DEFINE CHAR $('a'..'z'|'A'..'Z')$;
    // helper token - not used on its own
    DEFINE FRAGMENT DIGIT $('0'..'9')$;
    // composed token
    DEFINE IDENTIFIER CHAR + $($ + CHAR + $|$ + DIGIT + $)*$;
}
```

3.2.3 Tokens Priorities

EMFText does automatically sort token definitions. However, sometimes token definitions might be ambiguous (i.e., the regular expressions defined for two different token are not disjoint). In such cases EMFText will always prefer the token defined first in the specification. By default, the predefined tokens (TEXT, WHITESPACE and LINEBREAK) have lower precedence than any explicitly defined token. However, they can be given a higher priority by prioritizing them over other tokens using the following directive:

```
TOKENS {
        PRIORITIZE NameOfPredefinedToken;
        DEFINE SOME_CUSTOM_TOKEN $someCustomRegularExpression$;
}
```

3.3 Token Styles

To define the default syntax highlighting for a language, a special section TOKENSTYLES can be used. For each token or keyword the color and style (BOLD, ITALIC, STRIKETHROUGH, UNDERLINE) can be defined as follows:

```
TOKENSTYLES {
    // show YOUR_TOKEN in black
    "YOUR_TOKEN" COLOR #000000;
    // show keyword 'public' in red and bold font face
    "public" COLOR #FF0000, BOLD;
}
```

The default highlighting can still be customized in runtime environments by using the generated preference pages.

3.4 Syntax Rules

For each concrete metaclass you can define a syntax rule. The rule specifies what the text that represents instances of the class looks like. Rule have two sides—a left and right-hand side. The left side denotes the name of the meta class, while the right-hand side defines the syntax elements.

3.4.1 Simple Syntax

The most basic form of a syntax rule is:

```
YourMetaClass ::= "someKeyword" ;
```

This rule states that whenever the text `someKeyword` is found, an instance of `YourMetaClass` must be created. Besides text elements that are expected "as is", parts of the syntax can be optional or repeating. For example the syntax rule:

```
YourMetaClassWithOptionalSyntax ::= ("#")? "someKeyword" ;
```

states that instances of `YourMetaClassWithOptionalSyntax` can be represented both by `#someKeyword` and `someKeyword`. Similar behavior can be defined using a star instead of a question mark. The syntax enclosed in the parenthesis can then be repeated. For example,

```
YourMetaClassWithRepeatingSyntax ::= ("#")* "someKeyword" ;
```

allows to represent instances of metaclass `YourMetaClassWithRepeatingSyntax` by writing `someKeyword`, `#someKeyword`, `##someKeyword`, or any other number of hash symbols followed by `someKeyword`. One can also use a plus sign instead of a star or question mark. In this case, the syntax enclosed in the parenthesis can be repeated, but must appear at least once.

3.4.2 Syntax for EAttributes

If metaclasses have attributes, we can also specify syntax for the value of these attributes. To do so, simply add brackets after the name of the attribute:

```
YourMetaClassWithAttribute ::= yourAttribute[] ;
```

Optionally one can specify the name of a token inside the brackets. For example:

```
YourMetaClassWithAttribute ::= yourAttribute[MY_TOKEN] ;
```

If the token name is omitted, as in the first example, EMFText uses the predefined token `TEXT`, which includes alphanumeric characters. The found text is automatically converted to the type of the attribute. If this conversion is not successfull, an error is raised when opening a file containing wrong syntax. For details on customizing the conversion of tokens, see Sect. 4.2.1.

Another possibility to specify the token definition that shall be used to match the text for the attribute value is do it inline. For example

```
YourMetaClassWithAttribute ::= yourAttribute['(',')'] ;
```

can be used to express that the text for the value if the attribute `yourAttribute` must be enclosed in parenthesis. Between the parenthesis arbitrary characters (except the closing parenthesis) are allowed. Other characters can be used as prefix and suffix here as well.

By default the suffix character (in the example above this was the closing parenthesis) can not be part of the text for the attribute value. To allow this, an escape character needs to be supplied:

```
YourMetaClassWithAttribute ::= yourAttribute['(',')','\'] ;
```

Here the backslash can be used inside the parenthesis to escape the closing parenthesis. It must then also be used to escape itself. That is, one must write two backslash characters to represent one.

To give an example on how escaping works, consider the following text: `(text(more\))`. After parsing, this yields the attribute value `text(more)`. The character sequence `\)` is replaced by `)`. Note that the opening parenthesis does not need to be escaped.

3.4.3 Syntax for EReferences

Metaclasses can have references and consequently there is a way to specify syntax for these. EMF distinguishes between *containment* and *non-containment* references. In an EMF model, the elements that are referenced with the former type are contained in the parent elements. EMFText thus expects the text for the contained elements (children) to be also contained in the parent's text.

The latter (non-containment) references are referenced only and are contained in another (parent) element. Thus, EMFText does not expect text that represents the referenced element, but a symbolic identifier that refers to the element. This is similar to the declaration and use of variables in Java. The declaration of a variable consists of the complete text that is required to describe a variable (e.g., its type). In contrast, when the variable is used at some other place it is simply referred to by its name. Non-containment references are similar to uses of variables.

Syntax for Containment References

A basic example for defining a rule for a meta class that has a containment reference looks like this:

```
YourContainerMetaClass ::= "CONTAINER" yourContainmentReference ;
```

It allows to represent instances of `YourContainerMetaClass` using the keyword `CONTAINER` followed by one instance of the type that `yourContainmentReference` points to. If multiple children need to be contained the following rule can be used:

```
YourContainerMetaClass ::= "CONTAINER" yourContainmentReference* ;
```

In addition, each containment reference can be restricted to allow only certain types, for example:

```
YourContainerMetaClass ::= "CONTAINER"
                           yourContainmentReference : SubClass ;
```

does allow only instances of `SubClass` after the keyword `CONTAINER` even though the reference `yourContainmentReference` may have a more general type. One can also add multiple subclass restrictions, which must then be separated by a comma:

```
YourContainerMetaClass ::= "CONTAINER"
                           yourContainmentReference : SubClassA, SubClassB ;
```

Syntax for Non-Containment References

A basic example for defining a rule for a metaclass that has a non-containment reference looks like this:

```
YourPointerMetaClass ::= "POINTER" yourNonContainmentReference[] ;
```

The rule is very similar to the one for containment references, but uses the additional brackets after the name of the reference. Within the brackets the token that the symbolic name must match can be defined. In the case above, the default token `TEXT` is used. Therefore, the syntax for an example instance of class `YourPointerMetaClass` can be `POINTER a`.

Since `a` is just a symbolic name that must be resolved to an actual model element, EMFText generates a Java class that resolves `a` to a target model element. This class be customized to specify how symbolic names are resolved to model elements. The default implementation of the resolver looks for all model elements that have the correct type (the type of `yourNonContainmentReference`) and that have a name or id attribute that matches the symbolic name. For details on how to customize the resolving of references, see Sect. 4.2.2.

3.4.4 Syntax for Printing Instructions

By default, EMFText can print all kinds of models. It does also preserve the layout of the textual representation when models are parsed and printed later on. However, to print models that have been created in memory, additional information can be passed to EMFText to cutomize the print result. This (optional) information includes the number of whitespaces and line breaks to be inserted between keywords, attribute values, references and contained elements. If you do not want to print models to text, printing instructions are not needed in your `.cs` file.

Syntax for Printing Whitespace

To explicitly print whitespace characters, the # operator can be used on the right side of syntax rules:

```
YourMetaclass ::= "keyword" #2 attribute[];
```

It is followed by a number that determines the number of whitespaces to be printed. In the example above, two whitespace characters are printed between the keyword and the attribute value.

Syntax for Printing Line Breaks

To explicitly print line breaks, the ! operator can be used on the right side of syntax rules:

```
YourMetaclass ::= "keyword" !0 attribute[];
```

It is followed by a number that determines the number of tab characters that shall be printed after the line break. In the example above, a line break is printed after `keyword`. The number of tabs refers to the current model element (i.e., `EObject`), which is printed.

3.4.5 Syntax for Expressions

To define syntax for metaclasses that represent expressions (e.g., binary expressions like additive or multiplicative expressions), one can use the `@Operator` annotation. This annotation can be added to all rules, which refer to expression metaclasses. For example, the rule:

```
@Operator(type="binary_left_associative", weight="1", superclass="Expression")
Additive ::= left "+" right;
```

can be used to define syntax for a metaclass `Additive`. The references `left` and `right` must be containment references and have type `Expression`, which is the abstract supertype for all metaclasses of the expression metamodel.

The `type` attribute specifies the kind of expression at hand, which can be binary (either `left_associative` or `right_associative`), `unary_prefix`, `unary_postfix` or `primitive`.

The `weight` attribute specifies the priority of one expression type over another. For example, if a second rule:

```
@Operator(type="binary_left_associative", weight="2", superclass="Expression")
Multiplicative ::= left "*" right;
```

is present, EMFText will create an expression tree, where `Multiplicative` nodes are created last (i.e., multiplicative expressions take precedence over additive expressions).

Unary expressions can be defined as follows:

```
@Operator(type="unary_prefix", weight="4", superclass="Expression")
Negation ::= "-" body;
```

There is also the option to define `unary_postfix` rules.

Primitive expressions can be defined as follows:

```
@Operator(type="primitive", weight="5", superclass="Expression")
IntegerLiteralExp ::= intValue[INTEGER_LITERAL];
```

They should be used for literals (e.g., numbers, constants or variables).

For examples how to use `@Operator` annotations see the SimpleMath language in the EMFText Syntax Zoo[1] and the ThreeValuedLogic DSL[2]. These do also come with an interpreter which shows how expression trees can be evaluated.

[1] `http://www.emftext.org/language/simplemath`
[2] `http://www.emftext.org/language/threevaluedlogic`

3.5 Suppressing warnings

To suppress warnings issued by EMFText in `.cs` files one can use the `@SuppressWarnings` annotation. This annotation can be added to rules, token definitions or complete syntax definitions. One can either suppress all warnings or just specific types. To suppress all warning for a syntax use the following syntax:

```
@SuppressWarnings
YourMetaClass ::= "someKeyword";
```

A list of all warning types can be found in Appendix **??**. For example, to suppress warnings about features without syntax, you may use:

```
@SuppressWarnings(featureWithoutSyntax)
YourMetaClassWithAttribute ::= "someKeyword";
```

4 DSL Customization

4.1 Customization Techniques

4.1.1 Overriding Generated Classes

4.1.2 Using Generated Extension Points

4.2 Concrete Customizations

4.2.1 Customizing Token Resolving

4.2.2 Customizing Reference Resolving

4.2.3 Implementing Post Processors

4.2.4 Implementing Quick Fixes

4.2.5 Implementing Builders

4.2.6 Implementing Interpreters

4.2.7 Customizing Text Hovers

4.2.8 Customizing Code Completion Proposals

List of Figures

List of Listings

A Code Generation Options

`additionalDependencies`

A list of comma separated plug-in IDs, which will be added to the manifest of the generated resource plug-in.

`additionalExports`

A list of comma separated packages, which will be added as exports to the manifest of the generated resource plug-in.

`additionalUIDependencies`

A list of comma separated plug-in IDs, which will be added to the manifest of the generated resource UI plug-in.

`additionalUIExports`

A list of comma separated packages, which will be added as exports to the manifest of the generated resource UI plug-in.

`antlrPluginID`

Sets the ID for the generated common ANTLR runtime plug-in.

`autofixSimpleLeftrecursion`

If set to true, EMFText will try to fix rules that contain simple left recursion. The default value for this option is **false**.

`backtracking`

If set to false, the Antlr-backtracking is deactivated for parser generation. The default value for this option is **true**.

`basePackage`

The name of the base package EMFText shall store the generated classes or the resource plug-in in.

`baseResourcePlugin`

The plug-in containing the resource implementation for the DSL (if different from the generated resource plug-in).

`defaultTokenName`

This option can be used to specify the name of the token that is used for non-containment references. A declaration like **featureX**[] in a CS rule with be replaced by **featureX**[**TOKEN_Y**] if the value of this option is **TOKEN_Y**.

`disableBuilder`

If set to true, the builder that is generated and registered by default will not be registered anymore. The default value for this option is **false**.

`disableEMFValidationConstraints`

If set to true, constraint validation using the EMF Validation Framework is disabled. The default value for this option is **false**.

`disableEValidators`

If set to false, constraint validation using registered EValidators will be enabled. The default value for this option is **true**.

`disableTokenSorting`

Disables the automatic sorting of tokens. The default value for this option is **false**.

`forceEOF`

If set to false, EMFText will generate a parser that does not expect an EOF signal at the end of the input stream. The default value for this option is **true**.

`generateCodeFromGeneratorModel`

If set to true, EMFText automatically generates the model code using the generator model referenced in the CS specification. The default value for this option is **false**.

`generateTestAction`

If set to true, EMFText generate a UI action that can be used to test parsing and printing of files containing textual syntax. The default value for this option is **false**.

`generateUIPlugin`

If set to false, EMFText will not generate the resource UI plug-in. The default value for this option is **true**.

`licenceHeader`

A URI pointing to a text file that contains a header which shall be added to all generated Java files.

`memoize`

If set to false, the Antlr-memoize is deactivated for parser generation. The default value for this option is **true**.

`overrideAbstractExpectedElement`

If set to false, the AbstractExpectedElement class will not be overridden. The default value for this option is **true**.

`overrideAbstractInterpreter`

If set to false, the AbstractInterpreter class will not be overridden. The default value for this option is **true**.

`overrideAdditionalExtensionParserExtensionPointSchema`

If set to false, the extension point schema for additional parsers is not overridden. The default value for this option is **true**.

`overrideAnnotationModel`

If set to false, the AnnotationModel class will not be overridden. The default value for this option is **true**.

`overrideAnnotationModelFactory`

If set to false, AnnotatioModelFactory class will not be overridden. The default value for this option is **true**.

`overrideAntlrPlugin`

If set to false, no ANTLR common runtime plug-in is generated. The default value for this option is **true**.

`overrideAntlrTokenHelper`

If set to false, the AntlrTokenHelper class will not be overridden. The default value for this option is **true**.

`overrideAttributeValueProvider`

If set to false, the AttributeValueProvider class will not be overridden. The default value for this option is **true**.

`overrideBracketInformationProvider`

If set to false, the BracketInformationProvider class will not be overridden. The default value for this option is **true**.

`overrideBracketPreferencePage`

If set to false, the BracketPreferencePage class will not be overridden. The default value for this option is **true**.

`overrideBracketSet`

If set to false, the BracketSet class will not be overridden. The default value for this option is **true**.

`overrideBrowserInformationControl`

If set to false, the BrowserInformationControl class will not be overridden. The default value for this option is **true**.

`overrideBuildProperties`

If set to false, the build.properties file will not be overridden. The default value for this option is **true**.

`overrideBuilder`

If set to false, the Builder class will not be overridden. The default value for this option is **true**.

`overrideBuilderAdapter`

If set to false, the BuilderAdapter class will not be overridden. The default value for this option is **true**.

`overrideCardinality`

If set to false, the Cardinality class will not be overridden. The default value for this option is **true**.

`overrideCastUtil`

If set to false, the CastUtil class will not be overridden. The default value for this option is **true**.

`overrideChoice`

If set to false, the Choice class will not be overridden. The default value for this option is **true**.

`overrideClasspath`

If set to false, the .classpath file of the resource plug-in will not be overridden. The default value for this option is **true**.

`overrideCodeCompletionHelper`
: If set to false, the CodeCompletionHelper class will not be overridden. The default value for this option is **true**.

`overrideCodeFoldingManager`
: If set to false, the CodeFoldingManager class will not be overridden. The default value for this option is **true**.

`overrideColorManager`
: If set to false, the ColorManager class will not be overridden. The default value for this option is **true**.

`overrideCompletionProcessor`
: If set to false, the CompletionProcessor class will not be overridden. The default value for this option is **true**.

`overrideCompletionProposal`
: If set to false, the CompletionProposal class will not be overridden. The default value for this option is **true**.

`overrideCompound`
: If set to false, the Compound class will not be overridden. The default value for this option is **true**.

`overrideContainment`
: If set to false, the Containment class will not be overridden. The default value for this option is **true**.

`overrideContextDependentURIFragment`
: If set to false, the ContextDependentUriFragment class will not be overridden. The default value for this option is **true**.

`overrideContextDependentURIFragmentFactory`
: If set to false, the ContextDependentUriFragmentFactory class will not be overridden. The default value for this option is **true**.

`overrideCopiedEList`
: If set to false, the CopiedEList class will not be overridden. The default value for this option is **true**.

`overrideCopiedEObjectInternalEList`
: If set to false, the CopiedEObjectInternalEList class will not be overridden. The default value for this option is **true**.

`overrideDefaultHoverTextProvider`
: If set to false, the DefaultHoverTextProvider class will not be overridden. The default value for this option is **true**.

`overrideDefaultLoadOptionsExtensionPointSchema`
: If set to false, the extension point schema for default load options is not overridden. The default value for this option is **true**.

`overrideDefaultResolverDelegate`

If set to false, the default resolver class will not be overridden. The default value for this option is **true**.

`overrideDefaultTokenResolver`

If set to false, the DefaultTokenResolver class will not be overridden. The default value for this option is **true**.

`overrideDelegatingResolveResult`

If set to false, the DelegatingResolveResult class will not be overridden. The default value for this option is **true**.

`overrideDocBrowserInformationControlInput`

If set to false, the DocBrowserInformationControlInput class will not be overridden. The default value for this option is **true**.

`overrideDummyEObject`

If set to false, the DummyEObject class will not be overridden. The default value for this option is **true**.

`overrideEClassUtil`

If set to false, the EClassUtil class will not be overridden. The default value for this option is **true**.

`overrideEObjectSelection`

If set to false, the EObjectSelection class will not be overridden. The default value for this option is **true**.

`overrideEObjectUtil`

If set to false, the EObjectUtil class will not be overridden. The default value for this option is **true**.

`overrideEProblemType`

If set to false, the EProblemType class will not be overridden. The default value for this option is **true**.

`overrideEditor`

If set to false, the Editor class will not be overridden. The default value for this option is **true**.

`overrideEditorConfiguration`

If set to false, the EditorConfiguration class will not be overridden. The default value for this option is **true**.

`overrideElementMapping`

If set to false, the ElementMapping class will not be overridden. The default value for this option is **true**.

`overrideExpectedCsString`

If set to false, the ExpectedCsString class will not be overridden. The default value for this option is **true**.

`overrideExpectedStructuralFeature`

If set to false, the ExpectedStructuralFeature class will not be overridden. The default value for this option is **true**.

`overrideExpectedTerminal`
: If set to false, the ExpectedTerminal class will not be overridden. The default value for this option is **true**.

`overrideFoldingInformationProvider`
: If set to false, the FoldingInformationProvider class will not be overridden. The default value for this option is **true**.

`overrideFollowSetProvider`
: If set to false, the FollowSetProvider class will not be overridden. The default value for this option is **true**.

`overrideFormattingElement`
: If set to false, the FormattingElement class will not be overridden. The default value for this option is **true**.

`overrideFuzzyResolveResult`
: If set to false, the FuzzyResolveResult class will not be overridden. The default value for this option is **true**.

`overrideGrammarInformationProvider`
: If set to false, the GrammarInformationProvider class will not be overridden. The default value for this option is **true**.

`overrideHTMLPrinter`
: If set to false, the HtmlPrinter class will not be overridden. The default value for this option is **true**.

`overrideHighlighting`
: If set to false, the Highlighting class will not be overridden. The default value for this option is **true**.

`overrideHoverTextProvider`
: If set to false, the HoverTextProvider class will not be overridden. The default value for this option is **true**.

`overrideHyperlink`
: If set to false, the Hyperlink class will not be overridden. The default value for this option is **true**.

`overrideHyperlinkDetector`
: If set to false, the HyperlinkDetector class will not be overridden. The default value for this option is **true**.

`overrideIBackgroundParsingListener`
: If set to false, the IBackgroundParsingListener class will not be overridden. The default value for this option is **true**.

`overrideIBracketHandler`
: If set to false, the IBracketHandler class will not be overridden. The default value for this option is **true**.

`overrideIBracketPair`

If set to false, the IBracketPair class will not be overridden. The default value for this option is **true**.

`overrideIBuilder`

If set to false, the IBuilder class will not be overridden. The default value for this option is **true**.

`overrideICommand`

If set to false, the ICommand class will not be overridden. The default value for this option is **true**.

`overrideIConfigurable`

If set to false, the IConfigurable class will not be overridden. The default value for this option is **true**.

`overrideIContextDependentURIFragment`

If set to false, the IContextDependentUriFragment class will not be overridden. The default value for this option is **true**.

`overrideIContextDependentURIFragmentFactory`

If set to false, the IContextDependentUriFragmentFactory class will not be overridden. The default value for this option is **true**.

`overrideIElementMapping`

If set to false, the IElementMapping class will not be overridden. The default value for this option is **true**.

`overrideIExpectedElement`

If set to false, the IExpectedElement class will not be overridden. The default value for this option is **true**.

`overrideIHoverTextProvider`

If set to false, the IHoverTextProvider class will not be overridden. The default value for this option is **true**.

`overrideIInputStreamProcessorProvider`

If set to false, the IInputStreamProcessorProvider class will not be overridden. The default value for this option is **true**.

`overrideILocationMap`

If set to false, the ILocationMap class will not be overridden. The default value for this option is **true**.

`overrideIMetaInformation`

If set to false, the IMetaInformation class will not be overridden. The default value for this option is **true**.

`overrideIOptionProvider`

If set to false, the IOptionProvider class will not be overridden. The default value for this option is **true**.

`overrideIOptions`

If set to false, the IOptions class will not be overridden. The default value for this option is **true**.

`overrideIParseResult`
: If set to false, the IParseResult class will not be overridden. The default value for this option is **true**.

`overrideIProblem`
: If set to false, the IProblem class will not be overridden. The default value for this option is **true**.

`overrideIQuickFix`
: If set to false, the IQuickFix class will not be overridden. The default value for this option is **true**.

`overrideIReferenceCache`
: If set to false, the IReferenceCache class will not be overridden. The default value for this option is **true**.

`overrideIReferenceMapping`
: If set to false, the IReferenceMapping class will not be overridden. The default value for this option is **true**.

`overrideIReferenceResolveResult`
: If set to false, the IReferenceResolveResult class will not be overridden. The default value for this option is **true**.

`overrideIReferenceResolver`
: If set to false, the IReferenceResolver class will not be overridden. The default value for this option is **true**.

`overrideIReferenceResolverSwitch`
: If set to false, the IReferenceResolverSwitch class will not be overridden. The default value for this option is **true**.

`overrideIResourcePostProcessor`
: If set to false, the IResourcePostProcessor class will not be overridden. The default value for this option is **true**.

`overrideIResourcePostProcessorProvider`
: If set to false, the IResourcePostProcessorProvider class will not be overridden. The default value for this option is **true**.

`overrideITextDiagnostic`
: If set to false, the ITextDiagnostic class will not be overridden. The default value for this option is **true**.

`overrideITextParser`
: If set to false, the ITextParser class will not be overridden. The default value for this option is **true**.

`overrideITextPrinter`
: If set to false, the ITextPrinter class will not be overridden. The default value for this option is **true**.

`overrideITextResource`

If set to false, the ITextResource class will not be overridden. The default value for this option is **true**.

`overrideITextResourcePluginPart`

If set to false, the ITextResourcePluginPart class will not be overridden. The default value for this option is **true**.

`overrideITextScanner`

If set to false, the ITextScanner class will not be overridden. The default value for this option is **true**.

`overrideITextToken`

If set to false, the ITextToken class will not be overridden. The default value for this option is **true**.

`overrideITokenResolveResult`

If set to false, the ITokenResolveResult class will not be overridden. The default value for this option is **true**.

`overrideITokenResolver`

If set to false, the ITokenResolver class will not be overridden. The default value for this option is **true**.

`overrideITokenResolverFactory`

If set to false, the ITokenResolverFactory class will not be overridden. The default value for this option is **true**.

`overrideITokenStyle`

If set to false, the ITokenStyle class will not be overridden. The default value for this option is **true**.

`overrideIURIMapping`

If set to false, the IUriMapping class will not be overridden. The default value for this option is **true**.

`overrideImageProvider`

If set to false, the ImageProvider class will not be overridden. The default value for this option is **true**.

`overrideInputStreamProcessor`

If set to false, the InputStreamProcessor class will not be overridden. The default value for this option is **true**.

`overrideKeyword`

If set to false, the Keyword class will not be overridden. The default value for this option is **true**.

`overrideLayoutInformation`

If set to false, the LayoutInformation class will not be overridden. The default value for this option is **true**.

`overrideLayoutInformationAdapter`

If set to false, the LayoutInformationAdapter class will not be overridden. The default value for this option is **true**.

`overrideLineBreak`
: If set to false, the LineBreak class will not be overridden. The default value for this option is **true**.

`overrideListUtil`
: If set to false, the ListUtil class will not be overridden. The default value for this option is **true**.

`overrideLocationMap`
: If set to false, the LocationMap class will not be overridden. The default value for this option is **true**.

`overrideManifest`
: If set to false, the manifest of the resource plug-in will not be overridden. The default value for this option is **true**.

`overrideMapUtil`
: If set to false, the MapUtil class will not be overridden. The default value for this option is **true**.

`overrideMarkerAnnotation`
: If set to false, the MarkerAnnotation class will not be overridden. The default value for this option is **true**.

`overrideMarkerHelper`
: If set to false, the MarkerHelper class will not be overridden. The default value for this option is **true**.

`overrideMarkerResolutionGenerator`
: If set to false, the MarkerResolutionGenerator class will not be overridden. The default value for this option is **true**.

`overrideMetaInformation`
: If set to false, the MetaInformation class will not be overridden. The default value for this option is **true**.

`overrideMinimalModelHelper`
: If set to false, the MinimalModelHelper class will not be overridden. The default value for this option is **true**.

`overrideNature`
: If set to false, the Nature class will not be overridden. The default value for this option is **true**.

`overrideNewFileContentProvider`
: If set to false, the NewFileContentProvider class will not be overridden. The default value for this option is **true**.

`overrideNewFileWizard`
: If set to false, the new file wizard class will not be overridden. The default value for this option is **true**.

`overrideNewFileWizardPage`

If set to false, the NewFileWizardPage class will not be overridden. The default value for this option is **true**.

`overrideOccurencePreferencePage`

If set to false, the OccurencePreferencePage class will not be overridden. The default value for this option is **true**.

`overrideOccurrence`

If set to false, the Occurence class will not be overridden. The default value for this option is **true**.

`overrideOutlinePage`

If set to false, the OutlinePage class will not be overridden. The default value for this option is **true**.

`overrideOutlinePageTreeViewer`

If set to false, the OutlinePageTreeViewer class will not be overridden. The default value for this option is **true**.

`overridePair`

If set to false, the Pair class will not be overridden. The default value for this option is **true**.

`overrideParseResult`

If set to false, the ParseResult class will not be overridden. The default value for this option is **true**.

`overrideParser`

If set to false, the Parser class will not be overridden. The default value for this option is **true**.

`overrideParsingStrategy`

If set to false, the ParsingStrategy class will not be overridden. The default value for this option is **true**.

`overridePixelConverter`

If set to false, the PixelConverter class will not be overridden. The default value for this option is **true**.

`overridePlaceholder`

If set to false, the Placeholder class will not be overridden. The default value for this option is **true**.

`overridePluginActivator`

If set to false, the PluginActivator class will not be overridden. The default value for this option is **true**.

`overridePluginXML`

If set to true, the plugin.xml file will be overridden. The default value for this option is **true**.

`overridePositionCategory`

If set to false, the PositionCategory class will not be overridden. The default value for this option is **true**.

`overridePositionHelper`
: If set to false, the PositionHelper class will not be overridden. The default value for this option is **true**.

`overridePreferenceConstants`
: If set to false, the PreferenceConstants class will not be overridden. The default value for this option is **true**.

`overridePreferenceInitializer`
: If set to false, the PreferenceInitializer class will not be overridden. The default value for this option is **true**.

`overridePreferencePage`
: If set to false, the PreferencePage class will not be overridden. The default value for this option is **true**.

`overridePrinter`
: If set to false, the printer will not be overridden. The default value for this option is **true**.

`overridePrinter2`
: If set to false, the Printer2 class will not be overridden. The default value for this option is **true**.

`overrideProblemClass`
: If set to false, the problem class will not be overridden. The default value for this option is **true**.

`overrideProjectFile`
: If set to false, the .project file of the resource plug-in will not be overridden. The default value for this option is **true**.

`overridePropertySheetPage`
: If set to false, the PropertySheetPage class will not be overridden. The default value for this option is **true**.

`overrideProposalPostProcessor`
: If set to false, the ProposalPostProcessor class will not be overridden. The default value for this option is **true**.

`overrideQuickAssistAssistant`
: If set to false, the QuickAssistAssistant class will not be overridden. The default value for this option is **true**.

`overrideQuickAssistProcessor`
: If set to false, the QuickAssistProcessor class will not be overridden. The default value for this option is **true**.

`overrideQuickFix`
: If set to false, the QuickFix class will not be overridden. The default value for this option is **true**.

`overrideReferenceResolveResult`

If set to false, the ReferenceResolveResult class will not be overridden. The default value for this option is **true**.

`overrideReferenceResolverSwitch`

If set to false, the reference resolver switch will not be overridden. The default value for this option is **true**.

`overrideReferenceResolvers`

If set to true, the reference resolver classes will be overridden. The default value for this option is **false**.

`overrideResourceFactory`

If set to false, the resource factory class will not be overridden. The default value for this option is **true**.

`overrideResourceFactoryDelegator`

If set to false, the ResourceFactoryDelegator class will not be overridden. The default value for this option is **true**.

`overrideResourceUtil`

If set to false, the ResourceUtil class will not be overridden. The default value for this option is **true**.

`overrideScanner`

If set to false, the Scanner class will not be overridden. The default value for this option is **true**.

`overrideSequence`

If set to false, the Sequence class will not be overridden. The default value for this option is **true**.

`overrideStreamUtil`

If set to false, the StreamUtil class will not be overridden. The default value for this option is **true**.

`overrideStringUtil`

If set to false, the StringUtil class will not be overridden. The default value for this option is **true**.

`overrideSyntaxColoringHelper`

If set to false, the SyntaxColoringHelper class will not be overridden. The default value for this option is **true**.

`overrideSyntaxColoringPreferencePage`

If set to false, the SyntaxColoringPreferencePage class will not be overridden. The default value for this option is **true**.

`overrideSyntaxCoverageInformationProvider`

If set to false, the SyntaxCoverageInformationProvider class will not be overridden. The default value for this option is **true**.

`overrideSyntaxElement`

If set to false, the SyntaxElement class will not be overridden. The default value for this option is **true**.

`overrideSyntaxElementDecorator`
: If set to false, the SyntaxElementDecorator class will not be overridden. The default value for this option is **true**.

`overrideTerminal`
: If set to false, the Terminal class will not be overridden. The default value for this option is **true**.

`overrideTerminateParsingException`
: If set to false, the TerminateParsingException class will not be overridden. The default value for this option is **true**.

`overrideTextHover`
: If set to false, the TextHover class will not be overridden. The default value for this option is **true**.

`overrideTextResource`
: If set to false, the text resource class will not be overridden. The default value for this option is **true**.

`overrideTextResourceUtil`
: If set to false, the TextResourceUtil class will not be overridden. The default value for this option is **true**.

`overrideTextToken`
: If set to false, the TextToken class will not be overridden. The default value for this option is **true**.

`overrideTokenResolveResult`
: If set to false, the TokenResolveResult class will not be overridden. The default value for this option is **true**.

`overrideTokenResolverFactory`
: If set to false, the token resolver factory class will not be overridden. The default value for this option is **true**.

`overrideTokenResolvers`
: If set to true, the token resolver classes will be overridden. The default value for this option is **false**.

`overrideTokenScanner`
: If set to false, the TokenScanner class will not be overridden. The default value for this option is **true**.

`overrideTokenStyleInformationProvider`
: If set to false, the TokenStyleInformationProvider class will not be overridden. The default value for this option is **true**.

`overrideUIBuildProperties`
: If set to false, the build.properties file of the resource UI plug-in will not be overridden. The default value for this option is **true**.

`overrideUIDotClasspath`

If set to false, the .classpath file of the resource UI plug-in will not be overridden. The default value for this option is **true**.

`overrideUIDotProject`

If set to false, the .project file of the resource UI plug-in will not be overridden. The default value for this option is **true**.

`overrideUIManifest`

If set to false, the manifest of the resource UI plug-in will not be overridden. The default value for this option is **true**.

`overrideUIMetaInformation`

If set to false, the MetaInformation class of the resource UI plug-in will not be overridden. The default value for this option is **true**.

`overrideUIPluginActivator`

If set to false, the plug-in activator class of the resource UI plug-in will not be overridden. The default value for this option is **true**.

`overrideUIPluginXML`

If set to false, the plugin.xml file of the resource UI plug-in will not be overridden. The default value for this option is **true**.

`overrideURIMapping`

If set to false, the UriMapping class will not be overridden. The default value for this option is **true**.

`overrideUnexpectedContentTypeException`

If set to false, the UnexpectedContentTypeException class will not be overridden. The default value for this option is **true**.

`overrideUnicodeConverter`

If set to false, the UnicodeConverter class will not be overridden. The default value for this option is **true**.

`overrideWhiteSpace`

If set to false, the WhiteSpace class will not be overridden. The default value for this option is **true**.

`parserGenerator`

The name of the parser generator to use.

`reloadGeneratorModel`

If set to true, EMFText reloads the generator model before loading it. This is particular useful, when the meta model (i.e., the Ecore file) is changing a lot during language development. The default value for this option is **false**.

`resourcePluginID`

The ID of the generated resource plug-in. The resource plug-in is stored in a folder that is equal to this ID.

`resourceUIPluginID`

The ID of the generated resource UI plug-in. The resource UI plug-in is stored in a folder that is equal to this ID.

`saveChangedResourcesOnly`
: If set to true, the generated EMF resource will save only resource when their content (text) has actually changed. The default value for this option is **false**.

`srcFolder`
: The name of the folder EMFText shall store the overridable classes of the resource plug-in in.

`srcGenFolder`
: The name of the folder EMFText shall store the generated classes of the resource plug-in in.

`tokenspace`
: The (numerical) value of this option defines how many whitespace should be printed between tokens if no whitespace information is given in CS rules. This option should only be used with the classic printer.

`uiBasePackage`
: The package where to store all clasess of the resource UI plug-in in.

`uiSrcFolder`
: The name of the folder EMFText shall store the overridable classes of the resource UI plug-in in.

`uiSrcGenFolder`
: The name of the folder EMFText shall store the generated classes of the resource UI plug-in in.

`useClassicPrinter`
: If set to false, the classic printer (i.e., the used before EMFText 1.3.0) will be used. Otherwise the new printer implementation is used. The default value for this option is **false**.

`usePredefinedTokens`
: If set to false, EMFText does not automatically provide predefined tokens (TEXT, WHITESPACE, LINEBREAK). The default value for this option is **true**.

B Types of Warnings

- abstractSyntaxHasStartSymbols
- collectInTokenUsedInRule
- duplicateOptionWithSameValue
- duplicateTokenStyle
- explicitSyntaxChoice
- featureWithoutSyntax
- generationWarning
- leftRecursiveRule
- licenceHeaderNotFound
- maxOccurenceMismatch
- minOccurenceMismatch
- multipleFeatureUse
- noRuleForMetaClass
- nonContainmentOpposite
- nonStandardOption
- oppositeFeatureWithoutSyntax
- optionalKeyword
- referenceToAbstractClassWithoutConcreteSubtypesInAbstractSyntax
- styleReferenceToNonExistingToken
- tokenOverlapping
- tokenPriorizationUselessWhenTokenSortingEnabled
- unusedResolverClass
- unusedToken