Grammars and meta-models

- Assignments are used to assign the parsed information to a feature of the current object.
- The type of the current object, its EClass, is specified by the return type of the parser rule.
- Example:

```java
State : 'state' name=ID ('actions' '{' (actions+=[Command])+ '}')? (transitions+=Transition)* 'end';
```

Attributes of State

Grammars and meta-models

- There are three different assignment operators, each with different semantics:
  - The simple equal sign '=' is the straightforward assignment, and used for features which take only one element.
  - The '+=' sign (the add operator) expects a multi-valued feature and adds the value on the right hand to that feature, which is a list feature.
  - The '?=' sign (boolean assignment operator) expects a feature of type EBoolean and sets it to true if the right hand side was consumed independently from the concrete value of the right hand side.

Grammars and meta-models

- Extended Backus-Naur Form Expressions
  - Token rules are described using "Extended Backus-Naur Form"-like (EBNF) expressions.
  - There are four different possible cardinalities:
    1. exactly one (the default, no operator)
    2. one or none (operator '?')
    3. any (zero or more, operator '*')
    4. one or more (operator '+')

Grammars and meta-models

- Unordered Groups
  - The elements of an unordered group can occur in any order but each element can occur at most once.
  - Unordered groups are separated with ' & ', e.g.
    ```java
    Modifier:
    static?='static' & final?='final' & visibility=Visibility;
    enum Visibility:
    PUBLIC='public' | PRIVATE='private' | PROTECTED='protected';
    ```
  - allows
    ```java
    public static final
    static protected
    final private static
    public
    ```
Grammars and meta-models

- Context-free grammars are mapped to signatures
- A signature describes the structure of abstract syntax trees
- A meta-model can also describe the structure of abstract syntax trees, plus
  - Relations between identifiers
  - Attributes to store scope information
  - Attributes to store type information

The Xtext specification for Booleans:

```plaintext
Model : OrBool ;
OrBool : lhs=AndBool ('|' rhs=OrBool)? ;
AndBool : lhs=NotBool ('&' rhs=AndBool)? ;
NotBool : (not? '=')? arg=BracketBool ;
BracketBool : '(' orArg=OrBool ')' | conArg=BoolCon ;
BoolCon : {TrueNode} 'true' | {FalseNode} 'false' ;
```

Resulting meta-model (of syntax tree) for Booleans:
Grammars and meta-models

First alternative for Boolean meta-model

SDF definition of toy language Pico

Xtext specification for Pico:

```plaintext
Model : Program ;
Program : 'begin' decls=Declss? stats=Statements? 'end' ;
Declss : 'declare' idtypes=IdTypes ';' ;
IdTypes : pairs+=IdType (',')* ;
IdType : name=ID ':' type=Type ;
Type : {naturalType} 'natural' | {stringType} 'string' ;
Statements : statements+=Statement (')' statements+=Statement)* ;
```
Grammars and meta-models

• Xtext specification for Pico (continued):

Statement : AssignStatement | IfStatement | WhileStatement ;

AssignStatement : lhs=ID ':=' rhs=Exp ;

IfStatement : 'if' Exp 'then' thenpart=Statements? 'else' elsepart=Statements? 'fi' ;

WhileStatement : 'while' Exp 'do' dopart=Statements? 'od' ;

Exp : lhs=Term (bop=BinOp rhs=Exp)? ;

Term : id=ID | literal=STRING | number=INT | '(' Exp ')' ;

BinOp: {plusOp} '+' | {minOp} '-' | {concOp} '||' ;

Resulting meta-model (of syntax tree) for Pico:

```plaintext
grammars org.example.pico with exp=expression.antlr.v2.antlr.Token

generate pico "http://www.xtext.org/example/pico"

public class pico {
    public static void main(String[] args) {
        String input = "(2 * 3) + 7";
        Number result = calculate(input);
        System.out.println("Result: " + result);
    }

    public static Number calculate(String expression) {
        // Implementation...
    }
}
```

Resulting meta-model (of syntax tree) for Pico:

```plaintext
platform source org.example.pico / src / main / java / org/example/pico/Parser.java

@package org.example.pico;
public class Parser {
    // Implementation...
}
```
Grammars and meta-models

• Resulting meta-model (of syntax tree) for Pico:

- Xtext specification for Pico (with cross references):
  ```plaintext
  AssignStatement : lhs=[IdType|ID] ':=' rhs=Exp ;
  Term : id=[IdType|ID] literal=STRING | number=INT | '(' Exp ')' ;
  ```

- Xtext offers
  • “built-in” cross reference mechanism
  • scoping mechanism via writing “simple” Java methods, see http://www.eclipse.org/Xtext/documentation/1.0.0/xtext.html#scoping

- Xtext mixes in fact context-free parsing with some form semantic evaluation
Grammars and meta-models

• Conclusions on Xtext
  • popular
  • well integrated in Eclipse
  • suited for defining concrete syntax of new languages
  • less suited for existing languages, because of LL class

• EMFtext
  • is tightly integrated with Eclipse Modeling Framework (EMF)
  • enables the definition of textual syntax for Ecore-based meta-models
  • offers a Concrete Syntax Specification Language (CS) is EBNF based
  • ANLTR based
  • Documentation: http://www.emftext.org/EMFTextGuide.php

Grammars and meta-models

• EMFtext offers
  • modular specification:
    - import mechanism for various meta-models
    - modularization and extension of CS specifications
  • default reference resolving mechanisms
    - default name resolution mechanism for models with globally unique names is available for any syntax
  • comprehensive syntax analysis
    - analyses of CS specifications inform the developer about potential errors

• Developing a language with EMFText is an iterative process and consists of the following basic tasks:
  1. specifying a language meta-model
  2. specifying the Concrete Syntax of the language
  3. generating language tooling
  4. optionally customizing the language tooling
Grammars and meta-models

- The meta-model (abstract syntax) of a language is specified using the Ecore Meta-modelling Language
- A CS specification consists of 4 sections:
  1. mandatory configuration:
     - the language file extension is defined
     - the syntax specification is bound to the meta-model
     - the syntax start symbol is defined
     - optionally, import of other syntaxes and meta-models and various EMFText code generation options can be configured
  2. basic token types used by the language lexer to tokenize language expressions are defined
  3. token styles are defined that customize syntax highlighting in the generated editor
  4. the syntax rules for the language are specified

Grammars and meta-models

- The syntax specification rules used in the CS language are derived from EBNF to support arbitrary context-free languages
  - to define syntax for EMF-based meta-models and relate to Ecore meta-modelling concepts
  - it provides Ecore-specific specializations of classical EBNF constructs like terminals, and non terminals
    - this enables EMFText to provide advanced support during syntax specification, e.g., errors and warnings if the syntax specification is inconsistent with the meta-model
    - it enables the EMFText parser generator to derive a parser that directly instantiates EMF models from language expressions

Grammars and meta-models

- Configuration section of CS:
  - First, the file extension used for the files, containing the models, must be defined via:
    SYNTAXDEF yourFileExtension
  - Second, the EMF generator model (.genmodel) containing the meta classes for which the syntax is specified. The genmodel can be referred to by its namespace URI:
    FOR <yourGenModelNamespaceURI> <yourGenmodelLocation>?
    - EMFText uses the generator model instead of the Ecore model, because it requires information about the code generated from the Ecore model

Grammars and meta-models

- Third, the start symbol must be defined, which must be a meta class from the meta-model:
  START YourRootMetaClassName
  - A CS specification can also have multiple start symbols (separated by a comma)
  - Typical candidates for start symbols are meta classes without incoming containment relations
Grammars and meta-models

• It is possible to import additional meta-models
  • if they are only referenced in the current one, and
  • a syntax for some or all of its concepts needs to be specified or reused

Meta-models and syntax specifications can be imported in a dedicated import section

IMPORTS {
  // imports go here
}

• There must be at least one import entry

Grammars and meta-models

• If a syntax is imported, all its rules are reused
  • Importing syntax rules is optional
  • One can also just import the meta-model contained in the generator model

prefix : <genModelURI> <locationOfTheGenmodel>

  // next line is optional
  WITH SYNTAX syntaxURI <locationOfTheSyntax>

Grammars and meta-models

• EMFText allows to specify custom tokens.
  • Each token consists of a name and a regular expression

By default, EMFText implicitly uses a set of predefined standard tokens:

TEXT : ('A'..'Z'|'a'..'z'|'0'..'9'|'_'|'')+
LINEBREAK : ('\r\n'|'\r'|'\n')
WHITESPACE : (' '|'	'|'')

• The predefined tokens can be explicitly excluded by using the usePredefinedTokens option:

OPTIONS {
  usePredefinedTokens
}

Grammars and meta-models

• A TOKENS section must be added to define:

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

  • Every token name starts with a capital letter
  • A regular expression must conform to the ANTLRv3 syntax for regular expressions (without semantic annotations)

• Example of composed tokens:

  TOKENS {
    DEFINE CHAR $('a'..'z'|'A'..'Z')$; // simple token
    DEFINE DIGIT $('0'..'9')$; // simple token
    DEFINE IDENTIFIER CHAR + $(CHAR + CHAR + $ | DIGIT + $)*$;
  }
Grammars and meta-models

• Syntax Rules
  • For each concrete meta class you can define a syntax rule
  • The rule specifies what the text that represents instances of the class looks like
  • Rules have two sides: a left and right-hand side.
    − the left side denotes the name of the meta class
    − the right-hand side defines the syntax elements

  • 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:
      YourMetaClassWithOptionalSyntax ::= (#)? "someKeyword" ;
      YourMetaClassWithRepeatingSyntax ::= (#)* "someKeyword" ;

  • If meta classes have attributes, we can also specify syntax for their values, by adding square brackets:
      YourMetaClassWithAttribute ::= yourAttribute[ ] ;
    • One can specify the name of a token inside the brackets:
      YourMetaClassWithAttribute ::= yourAttribute[MY_TOKEN] ;
    • If the token name is omitted EMFText uses the predefined token TEXT, which includes alphanumeric characters
      YourMetaClassWithAttribute ::= yourAttribute["set":["set"] ;

    • For boolean attributes, EMFText provides a special feature to ease syntax specification
      YourMetaClassWithAttribute ::= yourAttribute["yes" : "no"] ;
      • This rule states that yes represents the true value and no represents false.
      • You can also use the empty string for one of the values:
        YourMetaClassWithAttribute ::= yourAttribute["set" : ""] ;
        • The attribute is set to false by default and set to true in the text set is found
Grammars and meta-models

- For enumeration attributes, EMFText does also provide a special feature to ease syntax specification.
  - For each literal of the enumeration, the corresponding string representation must be given:

    ```
    YourMetaClassWithAttribute ::= yourAttribute[red : "r", green : "g", blue : "b"];
    ```

- Meta classes can have references and consequently there is a way to specify syntax for these.
  - EMF distinguishes between containment and non-containment references.
    - In EMF, the elements that are referenced with the former type are contained in the parent elements.
      - EMFText 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.
      - EMFText does not expect text that represents the referenced element, but a symbolic identifier that refers to the element.

- Each containment reference can be restricted to allow only certain types:

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

- 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:

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

    - Multiple subclass restrictions are also possible, separated by a comma:

      ```
      YourContainerMetaClass ::= "CONTAINER" yourContainmentReference : SubClassA, SubClassB ;
      ```
Grammars and meta-models

• An example for defining a rule for a meta class that has a non-containment reference looks like:

   YourPointerMetaClass ::= "POINTER" yourNonContainmentReference[] ;

• The rule is very similar to the one for containment references, but uses the additional brackets.

• Within the brackets the token that the symbolic name must match can be defined.

• Defining syntax for an expression language (e.g., arithmetic expressions) via EMFText can lead to:
  • structures that are not be optimal
  • left recursive rules

EMFText provides a special feature called operator precedence annotations (@Operator)

• These annotations can be added to rules, which refer to expression meta classes with a common superclass:

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

• defines syntax for a meta class Additive

• The references left and right must be containment references and have the 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:

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

• EMFText will create an expression tree, where Multiplicative nodes are created last (i.e., multiplicative expressions take precedence over additive expressions).
Grammars and meta-models

- **Unary expressions can be defined as follows:**
  
  ```
  @Operator(type="unary_prefix", weight="4", superclass="Expression")
  Negation ::= "-" body;
  ```

- **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)

**EMFtext offers:**
- concise definition of lexical and concrete syntax rules
- modularity

**EMFtext allows:**
- definition of editor features (syntax highlighting)
- pretty printing
- etc.