Cheat sheet on Eclipse Xtext

by Mikhail Barash
http://dsl-course.org

Based on L. Bettini's book *Implementing domain-specific languages with Xtext and Xtend* and Xtext Documentation

Terminal rules

```
terminal INT returns ecore::EInt : ('0'..'9')+ ;
```

terminal rule INT returns instances of `ecore::INT`

returns ecore::EInt — return type; by default, `ecore::EString`; any type can be returned, provided it is an instance of `ecore::EDataType`

'0'..'9' — character range; *character ranges are only available in terminal rules*

+ — cardinality

```
terminal DOUBLE : INT '.' INT ;
```

INT — rule call; within terminal rules, ***rules calls*** can only point to terminal rules

```
terminal IF_KEYWORD : 'if' ;
```

'if' — keyword; keywords can have any length and can contain arbitrary characters (including `\n`, `\r`, `\t`, `\b`, `\f`, and `\u123` for Unicode characters)

```
terminal FOO : 'f' . 'o' ;
```

. — wildcard; any sequence of characters; examples: `foo`, `fo`, `ff12345oo`; *wildcards are only available in terminal rules*

```
terminal MULTILINE_COMMENT : '/*' -> '*/' ;
```

-> — until token; everything should be consumed until a certain token occurs; *until tokens are only available in terminal rules*

```
terminal BETWEEN_HASHES : '#' (!'#')* '#' ;
```

!'#' — negated token; "inversion" of tokens (example: "not-hash"); *negated tokens are only available in terminal rules*

```
terminal ASCII : '0x' ('0'..'7') ('0'..'9' | 'A'..'F') ;
```

group; ('0'..'9' | 'A'..'F') — alternatives

Extended Backus-Naur Form Expressions

	cardinality	
`something`	[1]	exactly one (default, no operator is used)
`something?`	[0..1]	zero or one
`something*`	[0..n]	zero or more
`something+`	[1..n]	one or more

```
('int' ID ':=' Expression)
```

'int' — keyword; ID — terminal rule call; ':=' — keyword; Expression — rule call

sample code: `int x := 2+2`

```
'int' ID (':=' Expression)?
```

optional part; cardinality [0..1]

```
('int' ID ',')*
```

can be repeated any number of times or can be omitted

sample code: `int x, int y, int z,`

```
('int' ID ',')+
```

cannot be omitted

Parser rules

when a variable is declared, its `name` should be an `IDentifier`

parser rule

```
Variable : 'var' name=ID ';' ;
```

Variable — language concept; 'var' — keyword; name — feature of concept; ID — *terminal rule call*; ';' — keyword

```
Variable : (isFinal ?= 'final')? 'var' name=ID ';' ;
```

a variable can be declared as final; cardinality [0..1]

isFinal — Boolean feature

?= — Boolean assignment operator; expects a feature of type `EBoolean` and sets it to true if the right side was consumed, independently on the concrete value of the right-hand side

```
Class : 'class' name=ID '{'
                            fields += Variable*
        '}'
;
```

a class can have several fields; cardinality [0..n]

fields — multi-valued feature

+= — add operator; adds the value on the right-hand side to the feature, which is a list feature

Enums

enum rule

```
enum Visibility : PUBLIC='public' | PRIVATE='private' | PROTECTED='protected' ;
```

default value; enum always has an implicit default value which corresponds to the first value

```
Variable : visibility=Visibility? typeName=('int'|'string') name=ID ';' ;
```

if visibility is omitted, value `PUBLIC` will be assumed

sample code: `protected string s;` `int x;` — assumed `public`

Unordered groups

unordered group

```
Modifier : static?='static'? & final?='final'? & visibility=Visibility ;
```

members of an unordered group can occur in any order, but each member must appear once

`static` modifier can be given or omitted, `final` modifier can be given or omitted, visibility modifier must always be given

sample of code (valid): `public static final`

sample of code (valid): `static protected`

sample of code (valid): `final private static`

sample of code (valid): `public`

sample of code (erroneous): `static final static` — `static` appears twice

sample of code (erroneous): `public static final private` — visibility modifier appears twice

sample of code (erroneous): `final` — visibility modifier is missing

Expressions grammar

for languages with Java-like expressions, consider using Xbase

invalid definition

```
Expression : left=Expression ('+'|'-'|'*'|'/') right=Expression ;
```

left recursion — first symbol of the rule refers to the rule itself; *not compatible with LL(*) grammars used by ANTLR*

reference to itself here is not forbidden, because it is not the first symbol of the rule

rules for operators with lower priorities are defined in terms of rules for operators with higher priorities

```
Expr : Or ;

Or returns Expr : And ({Or.left=current} '||' right=And)* ;

And returns Expr : Equality ({And.left=current} '&&' right=Equality)* ;

Equality returns Expr : Comparison ({Equality.left=current} op=('=='|'!=') right=Comparison)* ;

Comparison returns Expr : PlusOrMinus ({Comparison.left=current} op=('>='|'<='|'>'|'<') right=PlusOrMinus)* ;

PlusOrMinus returns Expr : MulOrDiv ({PlusOrMinus.left=current} op=('+'|'-') right=MulOrDiv)* ;

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

zero or more — each `(...)*` group

as if `left`=`And` would be here

as if `left`=`Equality` would be here

as if `left`=`Comparison` would be here

as if `left`=`PlusOrMinus` would be here

as if `left`=`MulOrDiv` would be here

as if `left`=`Primary` would be here

```
Primary returns Expr :
      '(' Expr ')' |
      {Not} '!' expression=Primary |
      Atomic
;
```

```
Atomic returns Expr :
      {IntConst} value=INT |
      {StringConst} value=STRING |
      {BoolConst} value=('true'|'false') |
      {VarRef} var=[Variable]
;
```

INT, STRING — *terminal rule call*; [Variable] — *cross-reference*

Cross-references

an already declared variable can be assigned an expression

```
Assignment : [Variable] '=' Expression ';' ;
```

cross-reference to an existing `Variable`

concept within square brackets does not refer to a rule, but rather to an `EClass` (which is a type and not a parser rule)

sample code: `var x; x=1; y=0;`

assignment to variable `x` is allowed because this variable has been declared

assignment to variable `y` is not allowed because it has not been declared

cross-reference will be resolved by searching in the program for an element of type `Variable` with the given name

in order for this to work, the referred element must have a feature called `name`

Ambiguities and syntactic predicates

```
Conditional :
   'if' '(' condition=Expr ')' expressionWhenTrue=Expr
   ( =>'else' expressionWhenFalse=Expr )? ;
```

else-branch is optional

=>'else' — syntactic predicate; if parser is at this particular decision point and doesn't know what to do, check whether else keyword is present: if it is, then take that branch directly without considering other options that would match the same token sequence

Priority of operations

from highest to lowest

1	`!`	Boolean negation
2	`*`, `/`	multiplication and division
3	`+`, `-`	addition and subtraction
4	`<`, `<=`, `>`, `>=`	comparison
5	`==`, `!=`	equality and non-equality
6	`&&`	Boolean And
7	`||`	Boolean Or