Tutorial scenario day one

Hands-on xText, the logo turtle language

This document indicates the steps to follow in order to reproduce the tutorial.

The solution of this tutorial is available in the folder *part1-grammarfirst-solution*.

Installation

- Install an Oracle Java JDK (minimum 8, max 10, Oracle is preferred due to support of openFX used for the day 2 tutorial)
- grab and unzip the latest dev version of GEMOC Studio http://download.eclipse.org/gemoc/packages/nightly/

Create project and basic grammar

Start the first Eclipse, we will call it the **Language Workbench**

- File → new → project → xtext project
  - project name: fr.inria.sed.logo.xtext
  - Name: fr.inria.sed.logo.xtext.Logo
  - Extensions: logo
- montrer la grammaire de base
- generate mwe2

Create a launch configuration that runs a new Eclipse similar to the current one but also including the plugins under development in the workspace:

- Run → Run configurations... → Right click on Eclipse Application → new configutation

Start the Second Eclipse, we will call it the **Modeling workbench**

Create a project with an example model:

- new → project
- new → file, use **.logo** as file extension.
- Observe the basic features of the editor: completion/outline/syntax error marker

Switch back to the **Language Workbench**

- show model/generated/ecore as tree view
- create aird (warning as xtext delete the content of the "generated" folder , do not create the aird in it and prefer model or directly the root of the project
  - right click on model → create representation, → Initialization from semantic resource → select the Logo.ecore file

- select "Design" representation
- create Entities in class diagram
- populate with ecore content
- show ecore model as diagram

Improving the grammar

- improve grammar, let's play grammar first: create a few rules that allows to parse something like:

```
left 45
forward 15
right 90
forward 100
```

reference documentation https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html

```
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

LogoProgram :
    {LogoProgram}
    instructions+=Instruction (  instructions+=Instruction)*
    ;

Instruction :
    Forward | Left | Right  ;

Forward:
    'forward' steps=EInt;

Left :
    {Left}
    'left' angle=EInt;

Right :
    {Right}
    'right' angle=EInt;

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

add Procedure declaration

```
Instruction :
    Forward | Left | Right | ProcDeclaration ;

ProcDeclaration :
    {ProcDeclaration}
    'to'
    name=EString

        (  args+=Parameter)*
        instructions+=Instruction (  instructions+=Instruction)*
    'end';

Parameter returns Parameter:
    {Parameter}
    ':'name=EString;
EString returns ecore::EString:
    STRING | ID;
```

add procedure call, ie. reference to a ProcDeclaration

```
Instruction :
    Forward | Left | Right | ProcDeclaration | ProcCall;
ProcCall :
    declaration=[ProcDeclaration|EString]
    '(' (actualArgs+=EInt)? ( "," actualArgs+=EInt)* ')'
    ;
```

Switch to the **Modeling Workbench**

Observe the completion at work

Switch back to the **Language Workbench**

Better validation

reference documentation: https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#validation

- open the **LogoValidator.xtend** file
- add some checker

```
    @Check
    def checkPassedParameters(ProcCall procCall){
        if(procCall.actualArgs.size != procCall.declaration.args.size){
            warning('invalid number of argument, (expecting
'+procCall.declaration.args.size+')',
                procCall,
                LogoPackage.Literals.PROC_CALL__ACTUAL_ARGS
            )
        }
    }
```

Provide quickfix

reference documentation: https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html#quick-fixes

Add quickfix

add this line in the mwe2 file (ine the language section

```
    // quickfix API
            fragment = ui.quickfix.QuickfixProviderFragment2 {}
```

Regenerate

in the project xxx.logo.xtext.ui open new file **LogoQuickfixProvider.xtend** and add the following:

```
import static extension org.eclipse.xtext.EcoreUtil2.*
```

```
    @Fix(Diagnostic.LINKING_DIAGNOSTIC)
    def void fixMissingProcDecl(Issue issue,
                                IssueResolutionAcceptor acceptor) {
        if (issue.message.contains("ProcDeclaration")) {
            createMissingProcDecl(issue, acceptor);
        }
    }

    private def createMissingProcDecl(Issue issue, IssueResolutionAcceptor acceptor) {
        acceptor.accept(issue,
            "Create missing procedure declaration",
            "Create a new empty procedure declaration at the beginning of the file",
            null, // no icon
            [ element, context |
                val root = element.getContainerOfType(typeof(LogoProgram))
                root.instructions.add(
                    0,
                    LogoFactory::eINSTANCE.createProcDeclaration() => [
                        name = context.xtextDocument.get(issue.offset,
                        issue.length)
                    ]
                )
            ]
        );
    }
```

Formatting

reference documentation: see https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#formatting

When testing you can observe that there is no line break.

Additionally, if you do a right click → source → format, everything goes on single line.

Let's provide some autoformat informations

add in the mwe2 file:

```
language = StandardLanguage {
            ...
            // formatter API
            fragment = formatting.Formatter2Fragment2 {}
}
```

launch mwe2 generate.

open and fill the newly created **xxx.logo.xtext.formatting2.LogoFormatter.xtend** file.

```
class LogoFormatter extends AbstractFormatter2 {

    @Inject extension LogoGrammarAccess

    def dispatch void format(LogoProgram logoProgram, extension IFormattableDocument
document) {
        for (instruction : logoProgram.instructions) {
            instruction.format
            instruction.append[setNewLines(1, 1, 2)]
        }
    }

    def dispatch void format(ProcDeclaration procDeclaration, extension
IFormattableDocument document) {
        val to = procDeclaration.regionFor.keyword("to")
        val end = procDeclaration.regionFor.keyword("end")
        if(procDeclaration.args.empty) {
            val declName = procDeclaration.regionFor.feature(LogoPackage.Literals
.PROC_DECLARATION__NAME).append[newLine]
            interior(declName, end)[indent]
        } else {
            for ( arg : procDeclaration.args) {
                arg.surround[oneSpace]
            }
            procDeclaration.args.last.append[newLine]
            interior(procDeclaration.args.last.regionFor.feature(LogoPackage.Literals
.PARAMETER__NAME), end)[indent]
        }
        for (instruction : procDeclaration.instructions) {
            instruction.format
            instruction.append[setNewLines(1, 1, 2)]
        }
    }

    def dispatch void format(Block block, extension IFormattableDocument document) {
        val open = block.regionFor.keyword("[")
        val close = block.regionFor.keyword("]")
        open.append[newLine]
        interior(open, close)[indent]
        for (instruction : block.instructions) {
            instruction.format
            instruction.append[setNewLines(1, 1, 2)]
        }
    }
```

Adding a new editor: the tree base editor

show open with → Sample Reflective Ecore editor

explain the tree view + property view.

label and icon customization: 2 alternatives:

- if grammar first, use Xtext label provider https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html#label-provider
- if MM first (and support of other editors), prefer to edit the label provider in the .edit project

Common traps: management of the containment in the ecore (othewise the file cannot be serialized/saved)

advanced property view can be developed using tabs (https://www.eclipse.org/articles/Article-Tabbed-Properties/tabbed_properties_view.html and https://news.obeo.fr/en/post/let-me-sirius-that-for-you-properties-view)

Expression grammar

reference documentation: https://www.eclipse.org/Xtext/documentation/307_special_languages.html#expressions

add expression to evaluate

for Left recursive grammar the keyword *current* might be useful, see Associativity section in https://www.eclipse.org/Xtext/documentation/307_special_languages.html#expressions

TODO: vérifier l'utilisation de - dan les valeurs xtext

NOTE pour MM first vs grammar first * permet de mieux controler les arbres d'héritage * car difficulté de "regrouper" les attributs dans les classes parentes (UnaryExpression, binaryExpression, contriol Structure

Test project

Useful for non regression and checking the features.

open the *xxx.logo.xtext.tests* and add some new test that checks a logo program.

Launch it (*Right click on the project → Run as → JUnit Plugin test*)

For a better coverage of the feature (ie. Formatter test, validation test, etc) have a look to the examples available in your eclipse *File → New → Examples... → XText examples*

ui tests (outline, content assist, etc) are in *xxx.logo.xtext.ui.tests*

Other cool feature of XText

autres truc cool à tester éventuellement sur certains languages: https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html

- rename refactoring https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html#refactoring
- project and file wizard https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html#projectwizard

```
language = StandardLanguage {
            ...
            projectWizard = {
                generate = true
            }
            fileWizard = {
                generate = true
            }
}
```

- code mining https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html#code-mining
- hyperlinking
- outline view and label provider (but it might be more productive to do it on the edit plugin when using model first approach
- content assist
- template proposal
- advanced syntax coloring (lexical and semantic)
- support for qualified name, add in the mwe2 :

```
language = StandardLanguage {
            ...
            qualifiedNamesProvider = {}
}
```

- support for outline labels, add in the mwe2 :

```
language = StandardLanguage {
            ...
            labelProvider = {
                generateStub = true
            }
}
```

then customize the stub

other cool support:

- import file
- scope

Basic code/doc generator

open *LogoGenerator.xtend* and use it to write an html file that contains a list of all procedures.

This can be used to write some kind of compiler for exemple.

Xtend supports a template syntax that is really convenient for writing strings. See https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html#templates

```
class LogoGenerator extends AbstractGenerator {
    override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext context) {
        val List<ProcDeclaration> allProcDecl = resource.allContents.
filter(ProcDeclaration).toList
        val content = '''
        <html>
            <body>
                List of procedures declared in «resource.URI.segments.last».
                «FOR procDecl : allProcDecl BEFORE '<UL>' AFTER '</UL>'»
                    <LI>«procDecl.name» («FOR arg : procDecl.args SEPARATOR ', '»
«arg.name»«ENDFOR»)</LI>
                «ENDFOR»
            </body>
        </html>
        '''
        fsa.generateFile(resource.URI.segments.last+"_summary.html", content)
    }
}
```

Conclusion

XText documentation is relatively good but often difficult to reproduce. This is mainly due to evolution in its api. It is sometime useful to install the reference example provided in eclipse and mimic it in order to make it work. *File → New → Examples... → XText examples*

Xtext is easy to use for "regular" languages.

Xtext eases the development of a large set of modern editor features.

It targets Eclipse IDE but also some other IDEs, including browser based editors (ex: Monaco).