Xtend 2 - Model Transformations

Xtend tutorials

The two Xtend tutorials are stand-alone tutorials. You do not need to do the Xtend I tutorial in order to perform the Xtend II tutorial. Ask your supervisor which tutorial suits you best.

This installment of our little series of tutorials will be all about Xtend, a programming language that looks very similar to Java, but which adds some very convenient features. Xtend code compiles to Java and and was developed using Xtext. In fact, once you gain experience working with Xtend you will probably appreciate the power of Xtext even more.

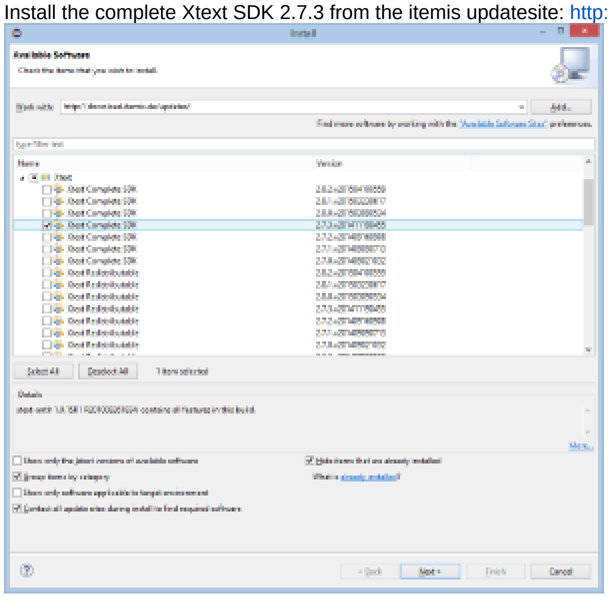
Preliminaries

There's a few things to do before we dive into the tutorial itself.

Required Software

For this tutorial, we need you to have Eclipse installed:

1. Install Eclipse. For what we do, we recommend installing the Eclipse Modeling Tools, with a few extras. Our Wiki page on getting Eclipse has the details: simply follow the instructions for downloading and installing Eclipse and you should be set.
2. Install the complete Xtext SDK 2.7.3 from the itemis updatesite: http://download.itemis.de/updates/ if you haven't done this already.



Recommended Tutorials

We recommend that you have completed the following tutorials before diving into this one.

1. Eclipse Plug-ins and Extension Points
2. Eclipse Modeling Framework (EMF)
    a. This tutorial needs the turingmachine.ecore and the controller you've implemented in the EMF tutorial. If you did not complete the EMF tutorial, you may download a working turing machine here... (in the future).

Required Tutorials

This tutorial continues from the Xtext 2 - Creating a Grammar from Scratch tutorial thematically. You might want to complete the Xtext I tutorial first.

Extended Brainfuck

Let's play a bit more with Brainfuck (BF). BF is nice, but not nice enough! Let's extend it by adding some syntactic sugar to the language. Feel free to express yourself here but you should at least add the following commands to eBF:

$	Increment the actual cell by 32
/	*Multiply the value of the actual cell by 2
a	*Copy the value of actual cell to the next cell on the right
A	*Copy the value in the next cell to the right to the actual cell
{ ... }	*Every command between { and the corresponding closing bracket } is duplicated immediately
0	*Set the actual cell to 0

*) may perform side effects on the tape (see below).

1. Open your BF grammar of the previous tutorial (or create a new one) and create rules so that the BF editor accepts the syntax changes.
2. Re-generate your BF code and run your eclipse instance to verify the changes.

Writing a Model Transformation

Now that you extended your BF syntax, it's time to start working on your model transformation. In particular, the first model-to-model (M2M) transformation you're going to write will transform an BF program into another BF program:

1. Add a new package `de.cau.cs.rtprak.<login>.brainfuck.transformations` to your BF project.
2. Add an *Xtend Class* to the new package.
3. If you notice that your new class is marked with an error marker because of a missing dependency of the new plug-in project to `org.eclipse.xtext.xbase.lib,` you can hover over the error with your mouse and have Eclipse add all libraries required by Xtend to your project.
4. Define an entry method for the transformation that takes a `BF program` instance as an argument and returns a BF `Program`. You can use the following (incomplete) method as a starting point:

```
 /**
 * Transforms a given Turing Machine into an imperative program model.
 *
 * @param machine the Turing Machine to transform into an imperative
 *                program.
 * @return a program model that implements the Turing Machine.
 */
def Brainfuck transformExtendedBrainfuck(Brainfuck brainfuck) {
    // Create the program we will transform the Turing Machine into
    val newBFProgram = BrainfuckFactory::eINSTANCE.createBrainfuck()

    // TODO: Your transformation code

    // Return the transformed program
    newBFProgram
}
```

   There's a few points to note here:

   - Lines in Xtend code don't have to and with a semicolon.
   - We have been explicit about the method's return type, but we could have easily omitted it, letting Xtend infer the return type.
   - The keyword `val` declares a constant, while `var` declares a variable. Try to make do with constants where possible.
   - The methods you call should be declared as `def private` since they are implementation details and shouldn't be called by other classes.
   - You may be tempted to add a few global variables that hold things like a global input variable or a pointer to the current state. While you could to that, `def create` methods might offer a better alternative...

5. Replace the `TODO` with an transformation code that takes an extended BF program and transforms it into an semantically equivalent BF program that only uses standard BF instructions.

HINT: Some of the extended BF commands can only be expressed by standard operations if you can write to other cells. Therefore you are allowed to perform side effects on the tape.

6. Open the *Plug-In Manifest Editor* and switch to the Runtime tab. Add the package containing your transformation to the list of exported packages. (You may have to check the *Show non-Java packages* option in the *Exported Packages* dialog to see the package.)

Testing your Model Transformation

You will need a way to test the transformation, so we will have to make it available through the UI. Eclipse plug-ins often come with a separate UI plug-in that contains the UI contributions, with the base plug-in only offering the functionality itself. In our case, a `brainfuck.ui` should already be present.

1. Go to your plugin to add a menu contribution. Open `plugin.xml` *Dependencies* and add org.eclipse.ui if not on the list yet. Go to *Extensions* and add org.eclipse.ui.menus.
   a. Add a *menuContribution* element to the new extension and set "popup:org.eclipse.ui.popup.any?after=additions" as *locationURI* (without quotation marks).
   b. Add a *command* element to the *menuContribution* with the following attributes:
      - *commandId:* de.cau.cs.rtprak.login.setSimFile
      - *label:* Set Simulation File
   c. Add a *visibleWhen* element to the *command*, and add an *iterate* element to the *visibleWhen* with the following attributes:
      - *operator:* and
      - *ifEmpty:* false
   d. Add an *adapt* element to the *iterate* with the following attribute:
      - *type:* org.eclipse.core.resources.IResource
   e. Add a *test* element to the *adapt* with the following attributes:
      - *property:* org.eclipse.core.resources.extension
      - *value:* brainfuck (or your file extension if you have chosen another one)
   f. Add a new extension org.eclipse.ui.commands and add a *command* element to it with the following attributes:
      - *id:* de.cau.cs.rtprak.login.setSimFile
      - *name:* Set Simulation File
      - Click on *defaultHandler* to open a dialog for creation of a new handler class. Name the new class `SetFileHandler` and put it into some package of that plugin. Remove the suggested interface and set `org.eclipse.core.commands.AbstractHandler` as superclass instead.

2. Create a command handler that loads the BF model from the selected file, calls the transformation on the model, and saves the newly generated BF program to a file with a different name (but same extension). You can use the following code as a template: (The code requires a dependency to `com.google.inject` to work.)

```java
@Override
public Object execute(ExecutionEvent event) throws ExecutionException {
    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (selection instanceof IStructuredSelection) {
        Object element = ((IStructuredSelection) selection).getFirstElement();
        if (element instanceof IFile) {
            IFile bfFile = (IFile) element;

            // Load Turing Machine
            Brainfuck bfProgram = loadBrainfuckProgram(bfFile);

            // Call the transformation
            Injector injector = Guice.createInjector();
            ExtendedBrainfuckTransformation transformation =
                    injector.getInstance(ExtendedBrainfuckTransformation.class);
            Program newBFProgram = transformation.transformExtendedBrainfuck(bfPRogram);

            // Save brainfuck program
            IFile newBFFile = bfFile.getParent().getFile(
                    new Path(bfFile.getName() + "2" + ".brainfuck"));
            saveBrainfuckProgram(newBFFile, newBFProgram);

            // Refresh the parent folder to have the new file show up in the UI
            try {
                bfFile.getParent().refreshLocal(IResource.DEPTH_ONE, null);
            } catch (CoreException e) {
                // Ignore
            }
        }
    }
    return null;
}

/**
 * Load the Brainfuck program model from the given file.
 *
 * @param bfFile the file to load the Brainfuck program model.
 * @return the Brainfuck model.
 * @throws ExecutionException if the file couldn't be opened.
 */
private brainfuck loadBrainfuckProgram(IFile bfFile) throws ExecutionException {
    // TODO Implement.
}

/**
 * Saves the given Brainfuck program.
 *
 * @param bfFile the Brainfuck file to save the program to.
 * @param bfProgram the program to save.
 * @throws ExecutionException if there was an error saving the file.
 */
private void saveBrainfuckProgram(IFile bfFile, Program bfProgram) throws ExecutionException {
    // TODO Implement
}
```