Code Generation with Xtend

Welcome to the second tutorial about Xtend. This time, you will take imperative language models and generate code for them in your favourite programming language. While we personally recommend Java, there's nothing stopping you from generating C code, PHP code, or anything else the constructs of our simple language model can be transformed into. We would just like to ask you to refrain from generating code in any esoteric programming language... 😉

The code generation you will implement here should not be optimized for generating the kinds of programs your transformation generates. Instead, we expect it to work for any valid program model. Further down, you will find an archive with example programs you can test your code generation on.

The slides to the presentation that accompanied this tutorial can be found here.

Contents

Preliminaries

You should have everything you need for this tutorial:

- The Turing Machine metamodel you created in the third tutorial.
- The Imperative Programming Language metamodel you downloaded in the previous tutorial.
- The Turing-Machine-to-Imperative-Programming-Language transformation you developed in the previous tutorial.

This tutorial will make xtendsive use of template expressions, so be sure to read up on those.

Generating Code with Xtend

We will of course need a new Xtend class that will take care of the code generation.

1. Add a new *Xtend Class* to the compiler project, preferrably in a new package called `de.cau.cs.rtprak.login.compiler.codegen`.
2. Add a method to your class that starts the code generation. It can look something like this:

```
/**
 * Generates Java code for the given imperative program.
 *
 * @param program the imperative program to generate code for.
 * @return code that implements the imperative program.
 */
def String generateCode(Program program) '''
    YOUR CODE GENERATION
'''
```

3. Decide, which programming language to generate code for. The easiest will probably be Java, but other languages should be fine too. Your code is supposed to generate code that is complete and compilable in your target language. Make sure that your code generation supports transformed Turing Machines as well as the sample programs we provide below.

Here's a few hints as to what your implementation should support:

- The imperative language supports arrays, which you could implement as lists or arrays. Either way, you should make sure that the generated code does not throw exceptions for reasonable array indices. Ideally, your code makes no assumptions on array sizes.
- You will have to generate some kind of a main method which takes care of initializing input variables, running the program, and printing the result. To initialize input variables, your generated code could ask the user for input on the console.
- Try to ensure that the generated code is readable and properly indented.

Making the Code Generation Available

As in the previous tutorial, add a menu contribution to the `...compiler.ui` plug-in to make the code generation available in the interface. Since your code generation implementation is expected to work for arbitrary (valid) instances of the programming language model, your menu contribution should be available for all programming language models and their textual representations. (".imperative" and ".pseudo" files) As in the previous tutorial, your handler should create a new file in the same directory as the input file and refresh the folder afterwards to have the file show up in the project explorer.

File Names

When working on your menu contribution, think about how to name the files the generated code will be stored in. If your target language is C, the file name doesn't matter much (as long as it has the .c file extension). If your target language happens to be Java, however, the file name makes quite a bit of a difference...

Testing Your Implementation

To test your implementation, you can use these sample programs. Since we haven't included comments in the textual syntax (or the model, for that matter), we'll just have to hope that the programs are self-explanatory...