5 Website Language

5.1 Language, Models and Grammar

In this Chapter, you will create your own grammar on the basis of some models of the language. With the help of these models, you will abstract the structure of the language into a grammar. The grammar will be used to describe a website. With its help, you should be able to specify the start page and other pages of a website including links to other pages and texts and pictures on pages. Additionally, the grammar should allow the specification of a navigation for the website. In combination with the generator that you will implement later on, it will give non-programmers the ability to create their own website by simply describing its structure.

The grammar has the name `Website` and extends only the MontiCore basic grammar `MCCommonLiterals`. Before you begin with writing the grammar for the Website language, look up this grammar to find out which nonterminals you can already use without declaring them yourselves. One of the models that we will provide you to create your grammar from is displayed in Listing 5.1. As you can see, a website will need a name, pages will need a name as well and you should be able to differentiate between the start page and other pages of the website. Pictures need a name as well. They are always in the format png to simplify things and their size must be specified. Links begin with the symbol -> followed by the name of the page that is referred and an optional title. Pages can have titles and always have a directory `content` in which text, links to other pages and pictures can be added in arbitrary order. You will find more models like this under `src/test/resources/de/monticore/gettingstarted/website`. In terms of symbol table we will design the language as simple as possible. As only pages can be referred to (by links or in the navigation), we will need PageSymbols to find the referred page with the help of the symbol table. No nonterminal necessarily needs to span a scope.

Exercise 1
The grammar `Website.mc4` already exists. Complete the grammar so that the language's parser can parse every website model into an AST. Use interface productions where useful and do not create nonterminals for things that are already defined in the "super" grammar like `Name`. Test your grammar by executing the tests in the class `ParseTest` which will try to use the parser of your grammar to parse the models. After you finished, ask your supervisor for our solution and compare it with yours.

5.2 CoCos

For some productions in this grammar, we will need several Context Conditions to ensure that the models are well-defined and that they make sense. First, we will need two CoCos that ensure that both the website name and every page name are capitalized. We need

```
website SERWTH {

  navigation {
    Home;
    Staff {
      Rumpe;
      Nagl;
    }
  }

  start page Home {          // start page
    // optional title not used
    content {
      pic Logo 167 x 40;     // file: Logo.png, scaled to 300x200px
      "Lorem Ipsum";         // text
      -> Staff;              // link to other page
      "Lorem ipsa";          // more text
    }
  }

  page Staff {
    content {                // arbitrary order
      -> Home;
      -> Rumpe "Prof. Dr. Rumpe";  // optional link title used
      -> Nagl  "Prof. Dr. Nagl";
    }
  }

  page Rumpe {
    "Prof. Dr. Rumpe";       // title used
    content {
      pic Rumpe 200 x 307;
      "Lorem ipsum";
    }
  }

  page Nagl {
    "Prof. Dr. Nagl";
    content {
      pic Nagl 190 x 285;
      "Lorem ipsum";
    }
  }

}
```

Listing 5.1: Example model for the language Website

another CoCo that checks that a website has exactly one start page. Additionally, the page names should be unique so that references to pages are unambiguous. Further, two CoCos have to be implemented to check that if a website or a link have a title, it should not be an empty string. Another CoCo should check that the width and height of a picture are not negative. Lastly, a page referenced by the navigation should exist. For this, we already created eight CoCo classes that are not implemented yet.

Exercise 2
Implement the eight CoCos whose skeletons we already created for you. Make certain that every CoCo implements the correct interface. Add them to the WebsiteCoCoChecker in the class `WebsiteCoCos` similar to the class `AutomataCoCos`. Test your CoCos by executing the tests in the `CoCoTest` class.

5.3 Generator

Our language is useful to describe a website with the help of a model. Why should we not write a generator that can generate websites we described in a model? This would make creating new websites easier for non-programmers and programmers alike. Non-programmers would not need to learn HTML to write their own website and could simply create a website by creating a model of the Website language and executing the generator to get their website. As the creation of a model and executing the generator is easier for a programmer than creating the whole website themselves with all this HTML code and the overhead of its tags, they could use the generator as well.

MontiCore provides the basis of our generator with its *GeneratorEngine*. It is able to process Freemarker-Templates (FTL) in combination with their arguments and the AST nodes of our language. Its main method is the method `generate` that needs to be given a (fully qualified) template name, an output file into which the template filled out with the correct arguments is generated, the AST node that the generated file corresponds to and a list of arguments of every type. Because we want to generate websites, we will generate one HTML file for every page of the website. Therefore, the ASTNode will be the ASTPage that is generated by the template. A template always has at least one argument called `ast`. This is the ASTNode that was passed to the `generate` method, in our case the ASTPage. Additional arguments that are given to the `generate` method must be declared in the template. For three additional arguments, the command could be `$tc.signature("a", "b", "c")` so that the variables `a`, `b` and `c` could be used in the template from now on. Using these arguments or using functions on these arguments can be done with the syntax `$expression`. The expression could be the argument itself like `$ast` or it could be a function of the argument like `$ast.getName()`. This is useful for the variable parts of the template. Listing 5.2 gives a short example on how to use it to set the title of a page in HTML.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>${ast.getName()}</title>
</head>
```

Listing 5.2: Setting the title of a website

In FTL, there are control structures such as `if`, `else` and `list`. The syntax for an if-statement is `<#if expression>`, where the expression should evaluate to a boolean. An else if has the syntax `<#elseif expression>`, an else has the syntax `<#else>`. After all elses and else ifs, the if-statement must be closed with an `</#if>`. Iterating through lists similar to a for each loop is done with `<#list navigationElementList as navElem>` where the navigationElementList may have the type `List<ASTNavigationElement>`. The variable `navElem` then has the type `ASTNavigationElement` and can be used like the variable of a for-each loop.

In programming languages, code that is reusable in different situations should be outsourced into a method or even a class. Similar to this, it may be more efficient or easier to outsource parts of the template (like the navigation) into other templates. To "call" this other template and pass necessary arguments to it like a method in Java, we use `$tc.include("path.to.Template", astnode, args...)`. The `astnode` could be of the type `ASTNavigation` or `ASTPage` depending on the context. It is also possible to give no additional arguments to the include-statement. For more information on using FTL, see https://freemarker.apache.org/.

Exercise 3

The skeleton of a generator for the Website language is already present in `src/main/-java/de/monticore/gettingstarted/website/WebsiteGenerator`. Complete the Java class with the help of MontiCore's GeneratorEngine and Freemarker Templates. It should generate an HTML file for every page of the website. The skeleton of the main template for a page can be found under `src/main/resources/website/Page.ftl`. Outsource parts of the template into other templates where useful. Tip: Use `website.Page.ftl` as the template name in the `generate` method of the GeneratorEngine. Your template should create a valid HTML website with a navigation (if present in the model), a title (if present in the model) and all contents of the page. Test your generator by executing the tests of the class `GeneratorTest` and looking at your generated websites.

The pictures that are referenced in the models can be found at `pics/modelName`. Copy the pictures for the correct model into the folder generated for the HTML files before viewing your website. If you do not do this, the pictures referenced in your HTML file cannot be found and therefore they will not be displayed on your website.

Exercise 4

Create a CSS file to prettify your website. By doing this, you will have to change your templates as well to add ids or classes to the HTML tags and to link your HTML files with the CSS file.

(Optional) Exercise 5

Design and generate your own website by creating a website model and executing the generator.

Exercise 6

Get ready to present everything you have learned in the Chapters for the language Website. Do this by either creating a short PowerPoint presentation or going through the exercises and presenting your solutions to the other group. The other group will also present their exercises to you.