Intelligent Editor Services for modular languages

Bachelor thesis

Jonathan Müller

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Computer Science
Prof. Dr. Dipl.-Ing. Mira Mezini

Thesis Statement pursuant to §22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. In the submitted thesis the written copies and the electronic version are identical in content.

Darmstadt, April 23th 2014 Jonathan Müller

Contents

Abstract

Language extension is a powerful tool for adding new syntactical constructs to a base language. SugarJ is an extensible programming language, which allows modular composition of sugar libraries, containing both syntax definitions and desugaring rules. Editor services have become a necessity for working with large software projects, but are often incapable of supporting modular extended languages. In this thesis, both a modular language based on JSP, HTML and JavaScript for the domain of dynamic web applications, as well as modular editor libraries to support editor services for the extended language are introduced. To provide editor services for modular languages, editor libraries are parsed at the same time as their dependent sourcefiles, which allows the incremental refining of a generic editor service of the Sugarclipse IDE.

1 Introduction

Embedding a Domain Specific Language (DSL) into any kind of host language is supposed to make the task of solving domain specific problems of that DSL easier.

There exist a number of ways to embed a DSL:

- A string-based embedding: The host language receives a program written in the DSL as a string and then parses and runs the code. A major drawback of this approach is the loss of static error-checking for the string, which, as far as the compiler of the host language is concerned, is syntactically correct. While the syntax of the DSL is practically fully embedded, it is difficult to compose it with other DSLs.

- Pure embedding as language constructs of the host language: That way only the semantics of the DSL are embedded with no regards to its syntax. It has been proposed to deploy pure embeddings as libraries.[9] A major advantage of this approach is the ease of composition, which is a direct result of only using the host language's syntax. Some languages, like Scala, partially address the problem of translating the DSL syntax into the host language syntax, by giving a certain degree of freedom in the form of overloading function calls.

- Modifications to the compiler of the host language: This allows for a true embedding of the DSL, both in a syntactical and semantical way. However, the workload of modifying the compiler might be not worth the value of an additional abstraction layer. Language composition would require further modifications to the compiler. An example of language extension via compiler modification would be Scala's support for XML, which has been directly embedded into the Scala compiler.[13]

An easier approach, retaining both the semantics and syntax of a DSL, as well as allows for modular composition via library import, is SugarJ[6]. It enables the usage of language libraries on a per-file basis. It will be shown how to extend the base language of SugarJ to both parse and compile a combination of JSP, HTML and an improved version of JavaScript.

Once a language has been extended, another problem arises. While a developer may now use the full extent of a DSL's specification, it is often the case that s/he does require editor support to fully make use of it. After all, the best way to learn about a language is to frequently use the auto-complete feature of an IDE. The best way to navigate large quantities of code is to use reference resolution and code outlining.

Editor services such as these are usually implemented for a fix set of languages, though. If the base language is extended, the editor services should grow with it. Adding editor service support to an IDE for a new or extended language requires a lot of effort.[3, 12, 11] The capabilities of Sugarclipse[5], an Eclipse version with editor support for SugarJ, will be explored in order to extend the base editor service simultaneously to extending the base language.

2 Background

2.1 Java Server Pages

JavaServer Pages (JSP) and Servlets are complementary technologies for producing dynamic web pages via Java. While Servlets are the foundation for server-side Java, they are not always the most efficient solution concerning development time. Coding, deploying, and debugging a Servlet can be a tedious task. Fixing a simple grammar or markup mistake requires wading through print() and println() calls, recompiling the Servlet and reloading a web application. Such a mistake easily happens. and the problem is compounded in complex Servlets. JSP is designed to complement Servlets by helping to solve this problem and simplifying Sevlet development.[7]

The syntax of JSP basically extends standard HTML with four new elements:

- <% .. %>- Skriptlet elements, which contain standard Java code
- <%= ... %>- Expression elements, which contain a single Java expression.
- <%@ ... %>- Directive elements, which contain meta-instructions for the JSP compiler (e.g. import instructions)
- <%! ... %>- Declaration elements, which add either Java variables, method declarations or class declarations to the Servlet upon compilation. Theoretically, other Java classes could access them as well, but practically, they are only used in a local context.

Since an aim of this thesis is to embed the modular language in Java as the host language, it is important to know, how JSP documents are translated into Servlets.

In Lst. 2.1 both the source code for a simple JSP document containing pure HTML and its counterpart after compilation into a Servlet can be seen. Because a Servlet will just print everything into its output, which is supposed to be interpreted on the client side of a web application, the Servlet contains only a number of println() statements with the pure HTML passing as their arguments.

Listing 2.1: Hello World as JSP and Servlet

```
<html>
  <head>
    <title>Hello, world</title>
  </head>
  <body>
    Hello, world
  </body>
</html>
```

```
// ...
public void doGet(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException{
  PrintWriter writer = response.getWriter();
  writer.println("<html>");
  writer.println("<head>");
  writer.println("<title>Hello, world</title>");
  writer.println("<body>");
  writer.println("Hello, world");
  writer.println("</body>");
  writer.println("</html>");
}
//...
```

For Java code in Skriptlet elements, which is interpreted on the server side, it is necessary to run the code in the Servlet's HTTP-methods. Java code is retained both semantically and syntactically after compiling the JSP document to a Servlet. Accordingly, the idea of using editor services designed towards Java on JSP documents is very plausible.

There are some differences between JSP-Java and pure Java, however. For example, instead of Java's import statement on the toplevel of a Java source file, JSP documents expect a directive element, containing the same fully qualified Java ID and the JSP import operator. Aside from imports, all kinds of Java toplevel declarations are prohibited in JSP, since there is no coherent way of translating them into Java Servlets.

The lifecycle of a JSP document starts with the compilation of the JSP document to a Servlet. Afterwards, the Servlet has to be deployed on a server, where it may handle HTTP requests. Incoming HTTP requests are handled at the runtime of the Servlet, and simultaneously a web page is created and sent as a HTTP response.

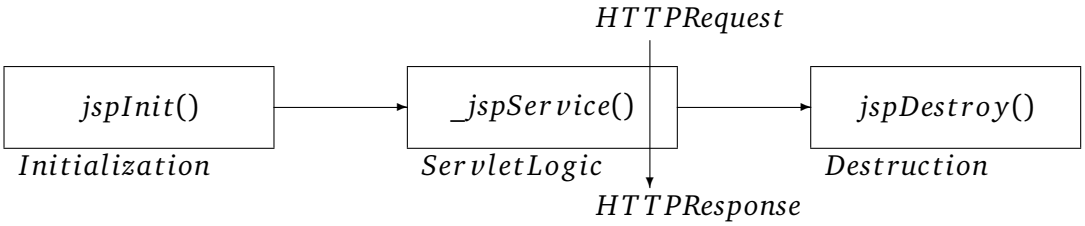

Figure 2.1: JSP Lifecycle

The method *_jspService*() takes on the roles of all of a Servlet's classical HTTP methods (e.g. doGet and doPost), which handle the different types of HTTP Requests. *jspInit()* and *jspDestroy()* each respectively fulfill the work of a Java constructor or destructor. It is possible to compile JSP documents in Servlets with classical HTTP methods, as is shown in 3.6.

Another way of creating dynamic content for web pages is to use the client-side interpreted language JavaScript. JavaScript is a language extension to HTML, which can access a site's *Document Object Model (DOM)* and which defines a set of keywords for event-handlers that can be used as HTML attributes.

2.2 SugarJ

Embedding DSLs is a problem which arises from the desire to express domain specific problems in their own language. To do so, the host language, in which the DSL is embedded, has to be able to run the DSL code. SugarJ[6] is an extensible programming language, allowing the organization of language extensions as *sugar libraries*, which can be imported and combined on a per-file basis.

Such libraries can contain code for a syntactic extension and code for its desugaring. Typically, the desugaring target language would be Java, the host language of SugarJ application code. Nevertheless, it is also possible to desugar into further syntactic extensions or desugarings.

Syntactic extension code is written in the Syntactic Definition Formalism language (SDF)[8]. The base grammar of SugarJ is a standard Java grammar, extended by toplevel sugar declarations. To write an extension for a DSL, it is sufficient to write down the DSL grammar in SDF inside a sugar library. To use the DSL syntax in a SugarJ program, it is necessary to import the extension as one would import a Java library.

Desugaring rules are expressed as transformations from abstract syntax trees to abstract syntax trees written in Stratego[2]. SugarJ complements those desugaring rules by allowing concrete Java syntax on either side of the transformation rule. Once all possible desugaring rules have been applied and, if the resulting abstract syntax tree can be pretty-printed as a valid Java program, it is pretty-printed.

The SugarJ compiler will incrementally compile files, one toplevel declaration at a time. This serves the purpose of modifying the grammar and desugarings, by using the user defined sugar libraries. That way, their content may influence both the parsing and the desugaring of the subsequent program.

Lst. 2.2 shows the sugar library code of the SugarJ case study for mathematical pairs[4], where the desugaring rules contain concrete Java syntax (enclosed by special parentheses |[and]| for concrete syntax) on the right. *pair.Pair* denotes a Java class, which uses generics to implement pairs semantically. The unary prefix operator ~ is used to insert Stratego terms into concrete Java syntax.

Listing 2.2: SugarJ Pairs

```
public extension PairExtension{
  context-free syntax
    "(" JavaType "," JavaType ")" → JavaType{cons("PType")}
    "(" JavaExpr "," JavaExpr ")" → JavaExpr{cons("PExpr")}

  desugarings
    desugar-pair-type
    desugar-pair-expr

  rules
    desugar-pair-type :
    PType(t1, t2) → |[ pair.Pair<~t1, ~t2> ]|

    desugar-pair-expr :
    PExpr(e1, e2) → |[ pair.Pair.create(~e1, ~e2) ]|
}
```

SugarJ's capability for language composition makes it a perfect candidate for modular languages. Ideally, all it takes is to import sugar libraries to compose multiple DSLs. However, ambiguities can arise. For example, since SDF does not have namespacing, it could prove necessary to manually rename overloaded non-literals.

2.3 Editor Services

To implement editor services, Sugarclipse [5] is utilized. It combines the sugar libraries of SugarJ[6] with the IDE foundation of Spoofax [10]. Like in SugarJ, it is possible to write editor libraries for Sugarclipse, which are parsed incrementally while extending the grammar of a generic editor component. This is done by adding a new compiler phase to the SugarJ compiler. After parsing a toplevel declaration, an AST will be created. The *analzye* phase will transform this AST into an equivalent AST with annotations; those annotations have no influence on the desugaring of the program. As with regular SugarJ, the desugared AST can then be used to generate the compiled base language.

Sugarclipse's editor services are imported like sugar libraries on a per-file basis. That way, it is possible to create editor libraries for multiple languages and combine them on a modular level. Usually, editor libraries are written for a specific DSL and will only affect the specific subtree of the AST, where that DSL will appear. That way, conflicts between different editor libraries for the same service should be avoided. If two libraries are describing different editor services altogether, there will not be any conflict at all.

Some conflicts cannot be avoided, though. Depending on the conflicting service, it may be resolved implicitly by either aggregating the results of different editor libraries (e.g. code completion), or by using a heuristic. In Sugarclipse, the closest-match heuristic will be used to decide, which result should be shown in the editor view.[5]

It has been proposed in the Sugarclipse paper[5] that editor libraries can be as easily extended and composed as sugar libraries, having similar restrictions regarding hostile environments.

3 Embedding JSP

3.1 Static HTML documents

3.1.1 Syntax for static HTML

To extend SugarJ's base language with HTML, a sugar library with the necessary grammar is required. In the spirit of modular languages, an existing sugar library for HTML will be used and modified. Its source code can be found at `https://github.com/sugar-lang/case-studies/blob/master/java-server-pages/src/html/HtmlSyntax.sugj`.

According to the W3C reference syntax for HTML [14], a HTML document requires the following elements to be considered valid:

- A DOCTYPE element, with a URL reference to the WC3 Syntax for HTML
- The *html* element, which encloses the rest of the document
- The *head* element, which must enclose the *title* element.
- The *body* element

Modern browsers are designed in an accommodating way, since they will accept invalid HTML documents, as well. Further they will, to a certain degree, render them in the desired way.

It is therefore a tradeoff between correctness and practicality, when formulating the grammar. In the HTML grammar used in this thesis, most invalid HTML documents are allowed. E.g.: the WC3 syntax requires attributes to be correct, which means they have to be in a set of allowed attribute names. This thesis' grammar allows every kind of attribute, as long as it is followed by a syntactically correct value.

Listing 3.1: Excerpt from HTML Syntax

```
context-free syntax
  Prologue Element Epilogue    →  Document{cons("Document")}
  DocTypeDeclaration?    →  Prologue
  HtmlComment*       →  Epilogue

  "<" ElemName Attribute* "/>"   →  Element{cons("EmptyElement")}
  "<" ElemName Attribute* ">" HtmlContent*
    "</" ElemName ">"    →  Element{cons("Element")}

  AttributeName "=" AttributeValue
            →  Attribute{cons("Attribute")}

  Element        →  HtmlContent
  HtmlComment      →  HtmlContent{cons("Comment")}
  CharData         →  HtmlContent{cons("HtmlText")}
```

HTML documents are structured like trees, meaning that some kind of "root" is required. In this case, it is the non-literal *Document*, for which the Stratego constructor with the label "Document" is called. While a constructor call is not required to parse a SugarJ file, it makes reading the AST easier by inserting nodes of the chosen label and allows for type matching during desugaring.

The non-literal Document is produced by the non-literals *Prologue*, *Element* and *Epilogue*. Prologue merely contains the Doctype declaration and is optional. Epilogue is optional, as well. It only serves to allow HTML comments at the bottom of the HTML document. Element is produced by either a regular HTML element, with an opening and a closing tag, attributes and content, or an empty element, which

only has one tag with attributes. The (optional) content of regular elements can be either text, HTML comments or nested elements.

One would expect elements to require an opening and a closing tag with the same name. SugarJ possesses the capability to add customized error messages during its compilation. By adding a sugar library, which contains a single Stratego error strategy, it is possible to match Element nodes and print an error to the Eclipse UI, whenever the two names of two Element nodes differ from each other.

Listing 3.2: HTML Error checking

```
public check Checks {
  errors
    Element(leftName, attributes, content, rightName) =
    (leftName, rightName, "HTML start and end tag need to coincide")
    where <not(structurally-equal)> (leftName, rightName)
}
```

Before the SugarJ compiler can parse documents containing this HTML grammar, it needs to be able to resolve it. In order to achieve this, the non-literals of the HTML grammar have to appear in the production of the parsed language, which, unless grammar files have already been loaded, is the base SugarJ language. JSP documents should be declared in a similar way as to how Java classes are declared. Since classes are *ToplevelDeclarations*, a production was written into this thesis' grammar, taking similar literals and non-literals as the production of a class. The keyword *class* was swapped for *jsp* and the non-literals of the class body were swapped for non-literals of the HTML grammar. As it can be seen in Lst. 3.3, the bridging grammar has been written extension to both base SugarJ and HTML by importing the grammar definitions, containing their respective non-literals.

Listing 3.3: JSP Base Grammar

```
import org.sugarj.languages.Java;
import html.HtmlSyntax;

public extension JspSyntax{
  context-free syntax
  AnnoOrExtensionMod* "jsp" JavaId JSPBody → ToplevelDeclaration{cons("JSPDec")}
  "{" JSPDoc "}" → JspBody{cons("JSPBody")}
  Content* → JSPDoc{cons("JSPDoc")}

  HtmlContent → Content{cons("HtmlContent")}
}
```

That way, the SugarJ compiler will be able to deduce that the previously unknown code it could not parse, is actually part of the HTML grammar. Lst. 3.4 shows a static HTML document and the resulting AST. *Content*, which is any kind of content to be expected in JSP documents, is distinguished from *HtmlContent*that consists of HTML's non-literals for elements, text, and so forth. That level of granularity has been added to allow for a more typesafe desugaring of any extended languages, making use of the HTML grammar. Also, the HTML syntax has been modified: originally, the non-literal HtmlContent was also named Content. However, since SugarJ does not have proper namespacing. Invalid JSP documents were allowed to be parsed (e.g. by allowing JSP's *Content* to appear within HTML elements).

Listing 3.4: Static HTML and AST

```
import jsp.JspSyntax;

public jsp StaticHTML{
  <html>
    <head></head>
    <body>foobar</body>
  </html>
}

CompilationUnit(
  [ TypeImportDec(TypeName(PackageOrTypeName(Id("jsp")), Id("Syntax"))),
    JSPDec(
          [Public()],
```

```
      Id("StaticHtml"),
      JSPBody(
        JSPDoc(
          [ HtmlContent(
            Element(
              ElemName("html"), [ ],
              [ Element(ElemName("head"), [], [], ElemName("head")),
                Element(ElemName("body"), [],
                [ Text([CharDataPart("foobar")])],
                ElemName("body")) ],
              ElemName("html"))
          )  ]
        )
    )
  ]
)
```

3.1.2 Desugaring for static HTML

To transform static HTML documents into Servlets, which generate exactly the same HTML document, a desugaring library was introduced. The desired result is a Servlet, containing a list of import statements and an annotated class with method declarations for a Servlet's HTTP-methods that, in turn, should contain a list of statements. In the scope of static HTML, those statements should exclusively be calls to *writer.println* and print the string representation of the HTML code to the HttpServletResponse argument of the method. The SugarJ compiler will create such a program, if the final transformed AST can be pretty-printed to a Java program. The goal is to transform a heterogenous AST in such a manner that

- The final AST describes a valid Java program
- All properties of a Servlet are achieved
- The syntactical and semantical properties of the static HTML document are not lost

The first applied desugaring rule, *start*, sets up the different import statements, which are required to compile a Servlet. It also delegates to *desugar-JSPDec*, which sets up the Servlet's class (including the method declaration for the unused *doPost*). *desugar-JSPBodyContent* will try to dispatch over a list of *HtmlContent* by calling *desugar-Html* on each item.

Finally, *desugar-Html* creates a list of Java statements, depending on what kind of HtmlContent it is applied on:

Listing 3.5: Desugaring rules for static HTML

```
desugar-Html:
  Element(ElemName(startname), attributes, content, ElemName(endname))
  →
  <concat>[ [ |[ writer.println("<" + ~startnameString + ">"); ]| ],
      desugaredContent,
      [ |[ writer.println("</" + ~endnameString + ">"); ]| ]
  ]
  where <map(desugar-Html)> content ⇒ desugaredContent
  ; <to-java-string> startname ⇒ startnameString
  ; <to-java-string> endname ⇒ endnameString

desugar-Html:
  EmptyElement(ElementName(name), attributes)
  →
  [ |[ writer.println("<" + ~nameString + "/>"); ]| ]
  where <to-java-string> name ⇒ nameString

desugar-Html:
  HtmlText(Text([CharDataPart(text)]))
  →
  [ |[ writer.println(~textString); ]| ]
```

```
  where <to-java-string> text ⇒ textString

desugar-Html:
  Comment(comment)
  →
  [ ]
```

desugar-Html is a recursive function, which applies itself on the content of an Element node. Desugaring a Comment creates an empty list, equivalent to no operation. This makes sense, because comments have no place in the compiled product. Since the AST only contains Stratego strings, they has to be transformed into Java literals before using them in the concrete Java syntax. This is done through the auxiliary rule *to-java-string*.

It is now possible to desugar the example from Lst. 3.4, which will compile to a valid Servlet that fulfills the criteria for the desired result:

Listing 3.6: Desugared Static HTML

```
import java.io.Exception;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/StaticHtml")
public class StaticHtml extends HttpServlet{
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException{
    PrintWriter writer = response.getWriter();
    writer.println("<" + "html" + ">");
    writer.println("<" + "head" + ">");
    writer.println("</" + "head" + ">");
    writer.println("<" + "body" + ">");
    writer.println("foobar");
    writer.println("</" + "body" + ">");
    writer.println("</" + "html" + ">");
  }

  protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException{ }
}
```

While the desugared Servlet is now valid, the concatenation of strings is done via the unperformant plus-operator. Instead, it would be possible to introduce a *StringBuffer*, which excels at concatenating large numbers of strings at runtime. This would require small modifications to the implementation of the rules *start* in order to import the class StringBuffer, as well as to *desugar-JSPBody* to create an instance of StringBuffer in the method body. Another possible solution would be to concatenate the strings directly during the desugaring phase. Stratego's API offers a strategy for concatenating strings. This might lead to easier-to-read compiled Servlets. An approach like this would require modifying the *desugar-Html* rule, to apply an auxiliary strategy for concatenating HTML strings before printing them.

Furthermore, if compared to a classical JSP Servlet, the compiled document does not possess the method *_jspService()* (see: Fig. 2.1), but instead classical implementations for a Servlet's HTTP methods.

The current desugaring rules can transform most static HTML documents. The desugaring rules for HTML attributes have yet to be added, finalizing the work on this sugar library. Attribute nodes appear inside Element and EmptyElement nodes, which in turn are desugared by *desugar-Html*. Attributes appear as a list and contain both a name and a value. Currently, the Stratego parameter *attributes* is ignored during the transformation. Now, the desugaring rules for Element and EmptyElement are modified, to print a Java string representation of their attributes.

Listing 3.7: Desugaring rules for HTML

```
desugar-Html:
```

```
  Element(ElemName(startname), attributes, content, ElemName(endname))
  →
  <concat>[ [ |[ writer.println("<" + ~startnameString + " " + ~attributesString + ">"); ]| ],
      desugaredContent,
      [ |[ writer.println("</" + ~endnameString + ">"); ]| ]
  ]
  where <map(desugar-Html)> content ⇒ desugaredContent
  ; <to-java-string> startname ⇒ startnameString
  ; <to-java-string> endname ⇒ endnameString
  ; <desugar-Attributes> attribtues ⇒ attributesString

desugar-Html:
  EmptyElement(ElementName(name), attributes)
  →
  [ |[ writer.println("<" + ~nameString + " " + ~attributesString + "/>"); ]| ]
  where <to-java-string> name ⇒ nameString
  ; <desugar-Attributes> attributes ⇒ attributesString

desugar-Attributes:
  [ ]
  →
  <to-java-string> ""

desugar-Attributes:
  [Attributes(AttrName(name), DoubleQuoted([CharDataPart(value)])) | tail]
  →
  Plus(
    Plus(
      Plus(
        Plus(
          <to-java-string> name,
          <to-java-string> "=\\\""
        ),
        <to-java-string> value
      ),
      <to-java-string> "\\\" "
    ),
    <desugar-Attributes> tail
  )
```

Java expressions can now be constructed, creating concatenated strings from Attribute nodes. Those expressions are in turn inserted inside the concrete Java syntax of desugar-Html, when printing the enclosing tags.

3.2 Dynamic Web Pages with Skriptlets

3.3 Syntax for JSP elements

Similar to the HTML syntax, this thesis' JSP sugar library built upon a preexisting sugar library for JSP Skriptlets(`https://github.com/sugar-lang/case-studies/blob/master/java-server-pages/src/jsp/JspSyntax.sugj`.

This sugar library has the basic grammar for JSP's additional elements already mapped out, with its desugaring yet to fill in. As it has already been mentioned, JSP is the extension of regular HTML with four new elements and their respective content.

The simplest of the elements is the expression element. It contains an expression of the JSP expression language and, when compiled to a Servlet, is replaced by the Servlet equivalent to the expression element. The latter is always a Java expression statement. So the grammar should allow expression elements to stand in every place, where the HTML non-literal HtmlContent is expected, and on its left-handed side it must require the literals for the tag as well as the non-literal for JSP expressions - which, in standard JSP, is the non-literal for a Java expression.

The directive element, which contains compiler directives for the JSP compiler, is similar in terms of its *depth* in complexity, but it allows a broader *width* of non-literals as its content in regular JSP. Each

directive attribute not only requires its own parsing, but also its own desugaring strategy. It may stand wherever any kind of JSP *Content* is expected. On its left side it requires the literals for a directive element, as well as an attribute. However, the current implementation only allows the attribute *import* and its value.

With the declaration element, it is possible to declare new class members inside the compiled Servlet. Those class members are equally as expressive as common Java class members and can be accessed in the same way.

The most interesting element is the Skriptlet element. Like the expression element, it may stand wherever HtmlContent is expected. Its content consists of plain Java statements. This makes sense, because a JSP compiler would copy those statements into the body of a method declaration. Those statements may also contain sequences of *escaped* HtmlContents. Those HtmlContents should be desugared just like any other kind of HtmlContent, but syntactically, they appear in a new place. This basically means, that the Skriptlet contains a list of Java statements, which in turn, are allowed to contain or be escaped HTML code.

To deal with ambiguities, restrictions on this particular part of the grammar must be placed:

- Since the literals for escaping HTML code are the same literals used for opening and closing Skriptlet elements, matching has to be restricted to the *longest-match* in case of an ambiguity. Otherwise, it is not easily decidable, whether the document contains either two Skriptlet elements, or one Skriptlet element with escaped HTML code inside.
- Also, since Skriptlet elements inside the escaped HTML of another Skriptlet element are not allowed, they are *rejected*.

The grammar for those four elements is contained in the same sugar library as the bridging grammar between base SugarJ and HTML, see Lst. 3.3.

Listing 3.8: JSP Base Grammar

```
//Expression element
"<%=" JSPExpression "%>" → HtmlContent{cons("JSPExpression")}
JavaExpr → JSPExpression

//Directive element
"<%@" "page" ImportDecl "%>" → Content{cons("JSPImport")}
"import=\"" ImportInformation "\"" → ImportDecl
JavaTypeName → ImportInformation{cons("ImportString")}

//Declaration element
"<%!" Declaration* "%>" → Content{cons("JSPDeclaration")}
JavaMethodDec   → Declaration
JavaClassDec    → Declaration
JavaFieldDec      → Declaration

//Skriptlet element
"<%" JSPSkriptletNotEscaped "%>" → HtmlContent{cons("JSPSkriptlet"), longest-match}
JavaStm* → JSPSkriptletNotEscaped{cons("JavaSkriptlet")}
"%>" HtmlContent* "<%" → JavaStm{cons("EscapedJSP"), longest-match}
"%>" Content* "<%" → JSPSkriptletNotEscaped{reject}
```

With this grammar in place, it is now possible to parse SugarJ files with the new JSP elements. As it was mentioned above, parsing a SugarJ file creates an abstract syntax tree, which can be transformed via Stratego rules. Those ASTs now can be even more heterogeneous, since they may contain nodes from SugarJ, HTML and the new JSP elements. An extension of the the desugaring rules for static HTML is needed, to cover those new nodes. Lst. 3.9 shows the source code of a JSP document with a Skriptlet element, as well as its AST, containing nodes from SugarJ's Java grammar, as intended to.

Listing 3.9: JSP Document with Skriptlet, Source and AST

```
import jsp.JspSyntax;

public jsp DynamicHtml{
<html>
  <head></head>
  <body>
    <% for(int i = 0; i < 3; i++){
      %> foobar <%
    } %>
  </body>
</html>
}
```

```
    TypeImportDec(TypeName(PackageOrTypeName(Id("jsp")), Id("Syntax"))),
    JSPDec(
          [Public()],
      Id("DynamicHtml"),
      JSPBody(
        JSPDoc(
          [ HtmlContent(
            Element(
              ElemName("html"), [ ],
              [ Element(ElemName("head"), [], [], ElemName("head")),
                Element(ElemName("body"), [],
                [ Skriptlet(
                JavaSkriptlet(
                  [ For(
                    LocalVarDec(
                      [], Int(), [VarDec(Id("i"), Lit(Deci("0")))]
                    ),
                    Some(Lt(ExprName(Id("i")), Lit(Deci("3")))),
                    [ PostIncr(ExprName(Id("i"))) ],
                    Block(
                      [EscapedJSP([HtmlText(Text([CharDataPart("foobar")]))])]
                    )
                    )]
                )
                  ) ],
              ElemName("body")) ],
              ElemName("html"))
          )  ]
        )
      )
    )
```

As expected, the AST now contains heterogeneous nodes from four different grammars (base SugarJ, HTML, JSP, the bridging grammar between HTML and JSP), which can also be intertwined. The *For* node, which is produced by the Java grammar, is inside the Skriptlet node, which comes from the JSP grammar, and contains the *EscapedJSP* node from the bridging grammar, which in turn contains the *Text* node from the HTML grammar.

3.3.1 Desugaring of JSP elements

Import statements are ToplevelDeclarations in Java, and because import statements in JSP are written as directive elements, it is necessary to *lift* those elements to proper ToplevelDeclarations. In this case, directive elements are enforced to appear only at the top of a JSP document by only giving a desugaring for that scenario, which usually is an unenforced convention. *JSPDec* nodes are matched on whether their *JSPBody* contains directive elements as the head of their body list. If so, an extraction strategy must be used upon the directive element, transforming it into an import statement and places it at the toplevel of the result tree. The tail of the body list is processed recursively, until no more directive elements are

contained. Since JSP documents are not written in the classical sense, however, import statements can be written just like in Java sourcefiles.

Listing 3.10: Desugaring rules for JSP Imports

```
desugar-JSPDec:
  JSPDec(mods, Id(name), JSPBody(JSPDoc([JSPImport(importType) | tail])))
  →
  [TypeImportDec(desugared-importType),
   <desugar-JSPDec>JSPDec(mods, Id(name), JSPBody(JSPDoc(tail))) ]
  where <desugar-JSPImport> importType ⇒ desugared-importType

desugar-JSPImport:
  ImportString(import)
  →
  import
```

With this, the type of the rule *desugar-JSPDec* gets the additional type *JSPDec -> List[ToplevelDeclaration]*. Desugared import statements are inserted below the sourcefiles own import statements and treated as such. This means that importing the same Java Type more than once will lead to a compiler error from the Java compiler, after the SugarJ file has been desugared.

In order to desugar declaration elements, the desugaring's result must be inserted inside the body of the Servlet's class. The class body is also created via the rule desugar-JSPDec, as soon as no more import statements have to be processed.

Listing 3.11: Desugaring rules for JSP Declarations

```
desugar-JSPDec:
  JSPDec(mods, Id(name), JSPBody(body))
  →
  ClassDec(
    ClassDecHead([ SingleElemAnno(TypeName(Id("WebServlet")), Lit(String([Chars(<concat-strings>["/",name])])))
          , Public() ],
        Id(name),
        None(),
        Some(SuperDec(ClassType(TypeName(Id("HttpServlet")), None()))),
        None()
    ),
    ClassBody(
      <concat>[ [ desugar-JSPBody>JSPBody(bodyWithoutDeclarations)
          , MethodDec(
            // Omitted: Method declaration for "doPost"
          ) ], desugaredDeclarations ]
    )
  )
  where <filter(?JSPDeclaration(_))> body ⇒ declarations
  ; <filter(not(?JSPDeclaration(_)))> body ⇒ bodyWithoutDeclarations
  ; <map(desugar-JSPDeclaration)> declarations ⇒ desugaredDeclarations

desugar-JSPDeclaration:
  JSPDeclaration(javaDeclaration)
  →
  javaDeclaration
```

For the body list, which contains only non-directive elements, the necessary method declaration for a Servlet's doGet method is constructed. The body content is desugared into a list of Java statements and inserted into the method body.

To do so, the remaining body list is processed recursively and the results are concatenated. That way, instead of creating a list of lists of Java statements, only a single, flat list is created. This is necessary, because EscapedJSP will always desugar into a list of Java statements.

To transform Skriptlet and Expression nodes into Java, it is necessary to write a strategy for each kind of possibly expected node. Those can be Elements, Comments, EmptyElements, as well as the JSP elements Skriptlet and Expression. There are already strategies for Elements, Comments and EmptyElements (Lst. 3.7).

- Previous *HtmlContent* should be desugared as before.

- Expression elements contain a single Java Expression, which should be evaluated by the Servlet at runtime and then printed to the outgoing HttpServletResponse.
- Skriptlet elements contain a list of both JavaStm and EscapedJSP nodes. A strategy, which can match to either, is mapped on the list and the result is concatenated, so its type is *List[JavaStm]*. EscapedJSP contains a list of HtmlContent, for which the strategy for desugaring HTML is used. Since each Java statement, which may contain nested Java statements, can also contain escaped code, they cannot be left untransformed. Instead, each of them requires an individual desugaring strategy of the type *JavaStm -> List[JavaStm]*. The transformation strategy for turning single Skriptlet *JavaStm* nodes into desugared Java statements can be used recursively on their nested statements.

The desugaring has been extended for static JSP documents by adding desugaring rules, compliant to above restrictions.

Listing 3.12: Desugaring rules for JSP Expressions and Skriptlets

```
desugar-Html:
  Expression(expr)
  →
  [ |[ writer.println(~expr); ]| ]

desugar-Html:
  Skriptlet(JavaSkriptlet(skriptletCode))
  →
  <concat>desugared-skriptletCode
  where <map(desugar-Skriptlet)> skriptletCode ⇒ desugared-skriptletCode

desugar-Skriptlet:
  If(condition, Block(ifBlock), Block(elseBlock))
  →
  [If(condition, Block(<concat>desugared-ifBlock), Block(<concat>desugared-elseBlock))]
  where <map(desugar-Skriptlet)> ifBlock ⇒ desugared-ifBlock
  ; <map(desugar-Skriptlet)> elseBlock ⇒ desugared-elseBlock

// Omitted: desugar-Skriptlet for If(condition, Block(ifBlock)), DoWhile, While, For, Try/Catch ...

desugar-Skriptlet:
  EscapedJSP(html)
  →
  <concat> desugaredHtml
  where <map(desugar-Html)> html ⇒ desugaredHtml
```

Since JSP Expressions are Java Expressions, they can be passed as an argument to *writer.println()*. Whenever a *Skriptlet* is encountered, the rule *desugar-Skriptlet* is called to recursively turn all occurrences of EscapedJSP into Java statements. It is now possible to desugar all four JSP elements into pure Java.

3.4 Dynamic Web Pages with JavaScript

3.4.1 Syntax for JavaScript

The JavaScript grammar built upon is taken from `https://github.com/sugar-lang/case-studies/blob/master/java-server-pages/src/javascript/JavaScriptSyntax.sugj`. The most complex part of the JavaScript syntax is the program syntax. Yet, the hooks to the HTML syntax are relatively simple. Whenever HtmlContent is expected, a *script* element with the attribute *language*="*javascript*" and a JavaScript program as its content may instead be expected. A JavaScript program may also be expected as the value of a *HtmlEventAttr*, the built-in attribute event handlers of JavaScript.

To bridge the grammars of HTML and JavaScript, a new sugar library has been implemented, introducing the *JavaScript* constructor to HtmlContet. It also eliminates the ambiguity occurring if one of

the keywords for event handlers had been used as an HTMLAttribute, by restricting HTMLAttributes to never be the keywords of event handlers. Otherwise, the parser would not know whether it found an HTMLAtribute or an event handler. This production can be seen in Lst. 3.13.

Listing 3.13: JavaScript Bridging Syntax

```
import javascript.JSSyntax;
import html.HtmlSyntax;

public extension HtmlWithJavascript {
  context-free syntax
    JavaScriptBlock → HtmlContent{cons("JavaScript")}
    "<" "script" "language" "=" "\"" "javascript" "\"" ">" JavaScriptProgram? "</" "script" ">" → JavaScriptBlock

  lexical syntax
    "script"     → ElementName {reject}
    EventHandlerName  → AttributeName {reject}
}
```

3.4.2 Desugaring for JavaScript

Like HTML, a Servlet will print JavaScript code to its outgoing HttpServletResponse, so desugaring of the AST for JavaScript programs into a list of calls to *write.println* is needed. As JavaScript is interpreted not as loosely as HTML, its grammar is more complex, but the desugaring follows a similar pattern:

- Construct string representations of all statements, declarations and expressions
- Retain both the syntax and semantics of the JavaScript code
- Print the string representations to the outgoing HTML document, while still fulfilling the criteria for JSP documents

Listing 3.14: JavaScript Desugaring

```
desugar-Html:
  JavaScript(None)
  →
  [ |[ writer.println("<script language=\"javascript\"></script>"); ]| ]

desugar-Html:
  JavaScript(Some(Program(javascriptProgram)))
  →
  <concat><map(desugar-JavaScript)> javascriptProgram

desugar-JavaScript:
  If(condition, statement)
  →
  <concat>  [ [ |[ writer.println("if("); ]| ],
        desugaredCondition,
        [ |[ writer.println(")"); ]| ],
        desugaredStatement
      ]
  where <desugar-JSExpression> condition ⇒ desugaredCondition
  ; <desugar-JavaScript> statement ⇒ desugaredStatement

  // Omitted: Desugaring of complete JavaScript, which covers all possible statements, declarations, expressions ...
```

The JavaScript syntax had to be modified, to include new Stratego constructors, because otherwise the desugaring for JavaScript expressions would have been non-deterministic. All in all, desugaring rules for the 200 productions of the JavaScript syntax have been implemented.

The desugaring rules for event handlers were omitted for now, but they are implemented as well. Those rules should be applied whenever a *JavaScript* node or *HtmlEvent* node is found in the AST, which in turn may appear everywhere, where this grammar allows HTMLContent or Attributes. Thus, new

declarations for the rules *desugar-Html* and *desugar-Attributes* from Lst. 3.7 were introduced. However, a problem for desugaring HTML Attributes arises. The values of JavaScript event handlers are full-fledged JavaScript programs, for which the desugaring rule *desugar-JavaScript* (see Lst. 3.14) has the type *Program -> List[JavaStm]*. Yet, the rule *desugar-Attributes* tries to desugar to JavaStringLiteral. As Stratego is able to double-dispatch, this behavior is allowed. The resulting type-mismatch has to be handled, otherwise the desugaring will fail.

At this point, two options on how to handle this type conflict exist:

1. Write a new rule *transform-Program-to-AttributeValue*, which allows to use *desugar-HtmlAttribute* as before.

2. Rewrite the rule *desugar-HtmlAttribute*, so its outgoing type is *List[JavaStm]* - the same type as *desugar-Program*.

Option 2. is the more sensible one. Not only is the implementation of the complex rule *transform-Program-to-AttributeValue*unnecessary, which otherwise would require rewriting the set of desugaring rules for JavaScript for a different type, but the desugaring rules for static HTML are sanitized. This makes extension easier for all possible language extensions, which extend upon the Attribute non-literal.

Listing 3.15: Modified desugaring rules

```
desugar-Html:
  Element(ElemName(startname), attributes, content, ElemName(endname))
  →
  <concat>  [ [ |[ writer.println("<" + ~startnameString + " "); ]| ],
        desugaredAttributes,
        [ |[ writer.println(">"); ]| ],
        desugaredContent,
        [ |[ writer.println("</" + ~endnameString + ">"); ]| ]
      ]
  where <concat><map(desugar-Html)> content ⇒ desugaredContent
  ; <to-java-string> startname ⇒ startnameString
  ; <to-java-string> endname ⇒ endnameString
  ; <map(desugar-Attribute)> attributes ⇒ desugaredAttributes

desugar-Html:
  EmptyElement(ElemName(name), attributes)
  →
  <concat>  [ [ |[ writer.println("<" + ~nameString + " "); ]| ],
        desugaredAttributes,
        [ |[ writer.println("/>"); ]| ]
      ]
  where <to-java-string> name ⇒ nameString
  ; <map(desugar-Attribute)> attributes ⇒ desugaredAttributes

desugar-Attribute:
  Attribute(AttrName(name), DoubleQuoted([CharDataPart(value)]))
  →
  [ |[ writer.println(~nameString + "=\"" + ~valueString + "\"" + " "); ]| ]
  where <to-java-string> name ⇒ nameString
  ; <to-java-string> value ⇒ valueString
```

This allows an easy implementation of desugaring rules for event handlers:

Listing 3.16: Desugaring rules for JavaScript event handlers

```
desugar-Attribute:
  JSAttribute(HtmlEventAttr(name), javascriptProgram)
  →
  <concat>  [ [ |[ writer.println(~nameString + "=\""); ]| ],
        <desugar-JavaScript> javascriptProgram,
        [ |[ writer.println("\""); ]|
      ]
  where <to-java-string> name ⇒ nameString
```

Theoretically, error checking for JavaScript could be implemented in order to ensure that no undeclared functions or variables are called. Nevertheless, in JavaScript, those declarations may be found in

an external source. This is beyond the capabilities of the system right now, but could be an interesting avenue for future work.

3.5 Improved JavaScript

To show how powerful SugarJ is, the JavaScript grammar was improved by introducing a new kind of expression: JavaScript may be used to generate HTML content and sometimes requires unparsed embedded strings, containing HTML code for its function calls. A new production for *HtmlExpressions*was introduced, containing the non-literal Element from the HTML grammar. HtmlExpressions are parsed by the SugarJ compiler and thus, are automatically error-checked for syntactical soundness.

Listing 3.17: Improved JavaScript Syntax and Desugaring

```
context-free syntax
  Element →  PrimaryExpr{cons("HtmlExpr")}

rules
  desugar-JSPrimaryExpression:
    HtmlExpr(element)
    →
    <concat>  [ [ |[ writer.println("\""); ]| ],
          <desugar-Html>element,
          [ |[ writer.println("\""); ]| ]
        ]
```

Since HtmlExpression is formed by pure HTML code, no new desugarings had to be added. The desugaring strategies, devised earlier for static HTML documents, can simply be used again. Only the additional transformation strategy between JavaScript and HTML had to be added, just as it was done for regular JavaScript. The desugared HTML code has to be placed in quotes, so the JavaScript program may be run by a browser. The following example shows the application of this improved language.

```
import jsp.JspSyntax;
import jsp.JspDesugaring;
import javascript.ImprovedSyntaxAndDesugaring;

<html>
  <head>
    <script language="javascript">
      var htmlElement = <b>foobar</b>;
    </script>
  </head>
  <body></body>
</html>
```

3.6 Possible Extension: CSS

A possible extension to the language would be Cascading Style Sheets (CSS), an embedded DSL in HTML[14]. Since CSS is interpreted by the client, the goal would be to pass the desugared CSS to the HttpServletResponse of the Servlet's doGet method. To achieve this, two steps are required:

- Defining a syntax for CSS
- Implementing a desugaring for CSS, which fulfills the set goal

A possible challenge for defining the syntax, is to create the correct bridging productions to the other syntaxes. For example, a CSS declaration could contain JSP expressions in place of a property or value. Usually, the CSS syntax does not include JSP expressions in its productions, so this improvement would be similar to the previous improvement to the JavaScript syntax.

Implementing CSS as an extension would allow the resulting language to cover the domain of web applications:

- HTML for document structure and displaying content
- JavaScript for client-side dynamic applications
- JSP for server-side dynamic applications
- CSS as a DSL for describing the layout of the document

4 Editor Services

Editor services for the sugar libraries can be implemented either subsequently or simultaneously to the sugar libraries. Simultaneous production offers the advantage of early access to editor services during the development of the sugar libraries. A subsequent approach allows to use non-literals and AST nodes freely, which should not change their names at this stage anymore. Sugarclipse[5] offers support for a multitude of editor services, of which editor libraries for *syntax coloring, code folding* and *outlining* will be introduced.

- *Syntax coloring* highlights, as the name suggests, syntactical fragments with different colors and fonts.
- *Code folding* allows collapsing code fragments. Eclipse, for example, offers this service for Java class declarations.
- *Outlining* shows the structure of a sourcefile as a tree-like structure with (meaningful) named nodes and allows fast navigation.

4.1 Services for static HTML

In order to introduce an editor library, first, all sugar libraries with the productions for all relevant AST nodes/non-literals have to be imported. For static HTML pages, those nodes are:

- Element and EmptyElement
- Comment
- Prologue and Epilogue
- Attribute and AttributeName
- DoubleQuotedAttributeValue

To implement code folding, it has to be decided on what to fold. HTML elements are a natural choice in this matter. To define code-folding for elements, a section of rules is declared, by using the Sugarclipse keyword *folding*, followed by a list of matching rules for non-literals and AST nodes . The rules are given in the form *non-literal.node-label* and may contain the wildcard symbol _ . The production of elements is defined as

```
"<" ElementName Attribute* ">" HtmlContent* "</" ElementName ">"  → Element{cons("Element")}
```

, so to match them, the following rule is implemented:

```
folding
  Element.Element
```

This will let Sugarclipse know that whenever the undesugared AST of a sourcefile contains a node produced by above grammar, it can collapse the code by hiding all child nodes within and displaying only the uppermost line of code of the node.

For syntax coloring, a similar approach is applied. Until a declaration of a syntax coloring matches a syntactical fragment, the IDE's default color and font are used. To introduce syntax coloring for HTML, a section of rules with the keyword *colorer*, is declared, containing coloring rules. Coloring rules require a matching rule as in code folding, as well as a color definition. It is important to note that for the matched nodes, only the literals on the left side of their production are colored, as there is currently no (working) support for recursive color rules in Sugarclipse.

```
import editor.Colors;

colorer
  Elementname : blue
  Element    : blue

  Attribute    : darkblue
  AttributeName : darkblue
  AttributeValue   : darkorange
  AttributeValueText   : darkorange

  HtmlText     : 0 0 0 italic
```

The above code shows, how coloring for elements and attributes is achieved. Only declaring *Element : blue* will color an Element's brackets blue, because they are the non-literals of the production. The name of the Element, however, is given as a non-literal. It has to be declared separately. For HtmlText, a black, italic font has been chosen, which is declared on-the-spot. To achieve a unifying look among different syntactical fragments, it is easier to define colors as strategies beforehand, e.g. *blue = 0 0 255*, which can be found in the imported *editor.Colors*.

Outlining requires the implementation of Sugarclipse's *outline* strategy, which is expected to be of the type *Tree[Node] -> Tree[Label]*. Using Spoofax's auxiliary strategy *simple-label-outline*, a higher-order strategy of the required type, the AST is traversed and a given strategy of the type *Node -> Lable* is applied, which creates the labels, shown in the outline view. The outline data is then written to the AST as annotations.

Hence, the strategy *outline* has to be declared as followed:

```
html-to-outline-label:
  Element(ElemName(name), _, _, _) → <concat-strings>[ "<", name, ">" ]

html-to-outline-label:
  EmptyElement(ElemName(name), _) → <concat-strings>[ "<", name, "/>" ]

jsp-to-outline-label = html-to-outline-label
```

The actual implementation of *outline* is contained in the editor library, which contains the outlining strategies for the uppermost nodes of parsed JSP documents, and not in the editor library for static HTML elements. A more elaborate label could be constructed at this point by modifying the rule *html-to-outline-label*, but for now the name of an Element is shown as its label in the outline view.

To use the editor services for static HTML, the library must be imported into the files.

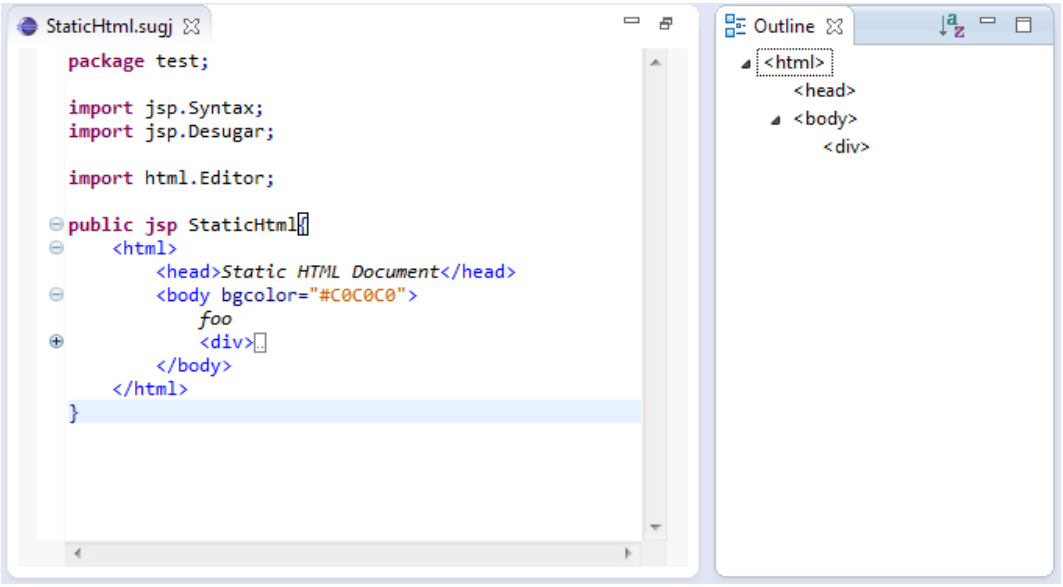

Figure 4.1: Editor Services for static HTML

4.2 Services for dynamic Web pages

Although the editor services are only defined for one language, they still work as expected for the extended language with JSP elements. Java code within JSP elements is already correctly colored and folded, because Sugarclipse delivers those services as a default for Java. Currently, Sugarclipse does not have a default outlining strategy for Java, though.

Similarly to the work done with HTML, code folding for JSP elements is achieved as followed:

1. Identify the syntactical fragments, ought to be folded
2. Look up the non-literals and nodes for those fragments in the grammar
3. Declare code folding for those non-literals and nodes

Listing 4.1: Editor Library for JSP elements

```
folding
  HtmlContent.JSPSkriptlet
  HtmlContent.JSPExpression
  Content.JSPImport
  Content.JSPDeclaration

colorer
  ToplevelDeclaration.JSPDec  : darkgreen
  JSPBody       : black
  Content.JSPImport     : keyword
  ImportString        : 0 0 0 italic
  HtmlContent.Skriptlet   : blue
  JavaStm.EscapedJSP    : blue

strategies
  outline = simple-label-outline(jsp-to-outline-label)

  jsp-to-outline-label:
    JSPDec(_, Id(name), _)  → name
```

Now, two editor libraries are established, which can be imported alongside. It must be remembered, however that the strategy *outline* is only implemented in the library for JSP elements. It would theoretically be possibleto implement the strategy outline in the editor library for HTML documents, as well, but the closest-match heuristic for editor services would make the behavior for an equally close match non-deterministic.

A document with both libraries imported, is colored/folded/outlined as expected:

Defining editor services for JavaScript first appears to be a tedious task, since the grammar is huge in comparison. However, JavaScript borrows multiple non-literals from the Java syntax, which in turn are colored correctly. Therefore, only syntax coloring for the bridging grammar between HTML and JavaScript has to be provided, as well as the usual folding and outlining declarations.

Listing 4.2: Editor Services for JavaScript

```
colorer
  HtmlContent.JavaScript    : red
  HtmlEventAttr     : grey
  Attribute.JSAttribute   : grey
  HtmlEventAttr.HtmlEventAttr : grey
  HtmlEvent       : grey

folding
  FunctionDec
  HtmlContent.JavaScript

strategies
  js-to-outline-label:
    FunctionDec(name, _, _) → name
  js-to-outline-label:
```

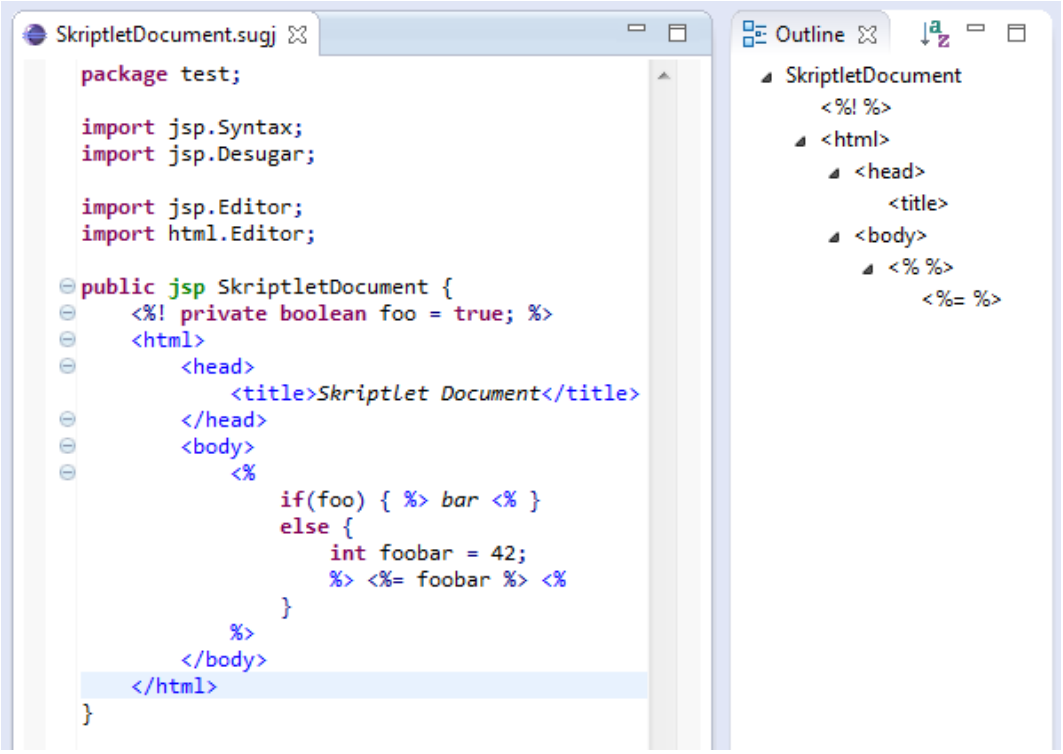

Figure 4.2: Editor Services for JSP

```
  JavaScript(_) → "JavaScript"

jsp-to-outline-label = js-to-outline-label
```

Should Java outlining ever be implemented in Sugarclipse, this would result in a working outlining strategy for JavaScript, as well. With the three editor libraries in place, even editor services for the improved JavaScript syntax can be offered, since no noteworthy non-literals have been introduced.

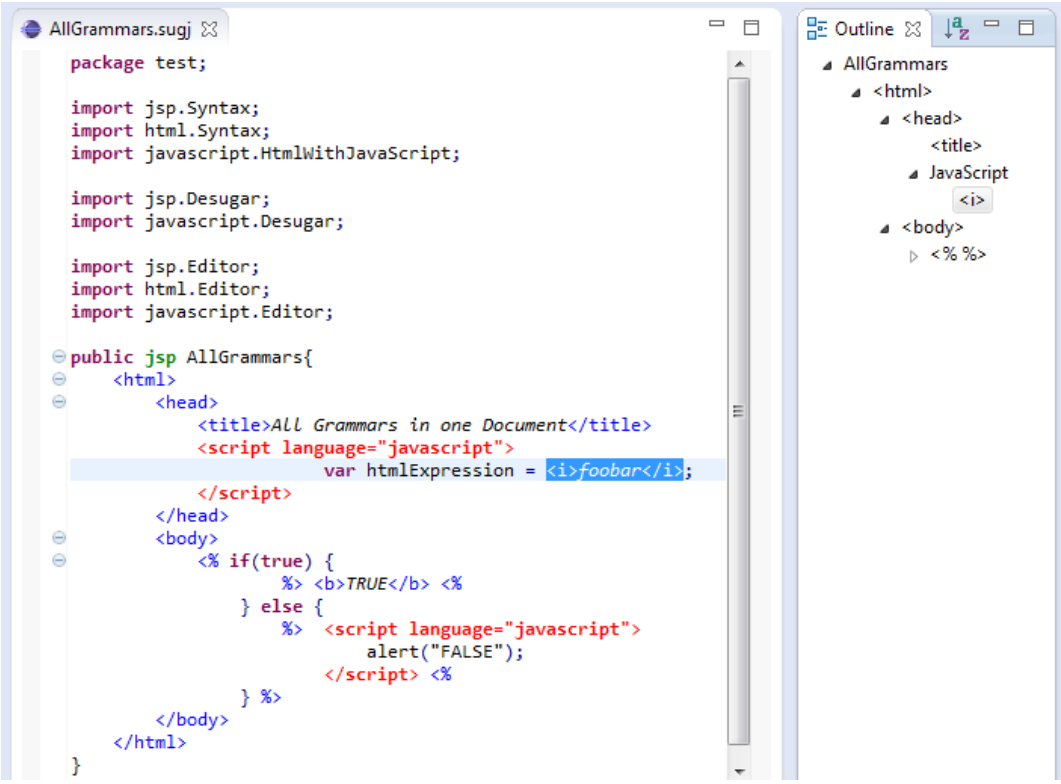

Figure 4.3: Editor Services for combined language

5 Related Work

The capability of extending editor libraries has been proven, alongside their respective language libraries. Now, this work has to be compared to other implementations of editor services.

5.1 Editor Services

Early adaptions of extensible editor services can be found with the editor *vim*. Its capability for syntax coloring is enhanced by plugins, written in script languages like vim script, and can contain language definitions and coloring schemes. Nevertheless, their capability for language extension is limited. It is possible to write scripts, which add the syntax of a host language to the syntax definition for an EDSL or vice versa, but composability is hard accomplish, since ultimately, those scripts do not abstract over the additional language syntaxes; instead they are designed to compose a specific set of languages. Plugins for convenient editor services, like code folding and outlining, are rare for those editors. Consequently, vim is unfit to navigate through large quantities of code.

In contrast, common IDEs, like Eclipse, IntelliJ, Netbeans or Visual Studio, provide the same extendibility via plugins, and usually support more editor services than vim and emacs. Compared to writing an editor library, developing a plugin is an immensely complex work.

A major drawback of a plugin-based approach, is its need to be designed for composition, whereas editor libraries inherently compose. Unlike editor libraries, plugins cannot easily be shipped alongside the product they were designed for. It would be imaginable, for example, to write an editor library with code completion, especially tailored for the usage with an accompanying API.

Currently, editor libraries are only working in Sugarclipse. Yet, in principal, they are agnostic about their IDE. As long as the IDE has a generic editor service, it should be possible to extend that editor service based on any editor library. This means, that language designers are capable of defining editor services for their language independently of any IDE.

5.2 Modular languages

Editor libraries are not the classical way of embedding DSLs. Instead, DSLs are often implemented either as string-based embeddings or as constructs of the host language.

In such a case, the DSL is part of the host langauge's syntax, which means that it adopts its editor services, as well. SugarJ parses the language on a finer level, than the standard Java compiler, as long as an appropriate language library is present. That allows for the use of editor libraries for the subset of the host language, which is the EDSL.

An example application would be Java's SQL API, JDBC, where SQL is embedded as pure strings.

This work focuses on SugarJ, whose host language is Java. However, a more refined Sugar-language called Sugar*[1] would allow extensions of any host language. Sugar* is based on SugarJ and offers the same capability for editor libraries.

Bibliography

[1] Sebastian Erdweg an Felix Rieger. A framework for extensible languages. *Proceedings of the 12th international conference on Generative programming: concepts & experiences*, pages 3–12, 2013.

[2] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72, 2005.

[3] M. Chapman. Extending JDT to support java-like languages. Invited Talk at EclipseCon '06, 2006.

[4] Sebastian Erdweg. Sugarj case study: Mathematical pairs, April 2014. URL `https://github.com/sugar-lang/case-studies/tree/master/pairs`.

[5] Sebastian Erdweg, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Growing a Language Environment with Editor Libraries. *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 167–176, 2011.

[6] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 391–406, 2011.

[7] Jayson Falkner and Kevin Jones. JavaServer pages. In *Servlets and JavaServer Pages*. Addison-Wesley, 2003.

[8] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf reference manual. *ACM SIGPLAN Notices*, 24:43–75.

[9] P. Hudak. Modular domain specific languages and tools. *Proceedings of International Conference on Software Reuse (ICSR)*, pages 134–142, 1998.

[10] Karl Trygve Kalleberg and Eelco Visser. Spoofax: An Extensible, Interactive Development Environment for Program Transformation with Stratego/XT. *TUD-SERG Technical Report Series*, (TUD-SERG-2007-018), 2007.

[11] C. Kästner, G. Saake, J. Feigenspann, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: Tool framework for feature-oriented software development. *Proceedings of International Conference on Software Engineering (ICSE)*, pages 611–614, 2009.

[12] S. McDirmid and M. Odersky. The scala plugin for eclipse. *Proceedings of Technology of Object-oriented Languages and Systems (TOOLS)*, pages 297–315, 2008.

[13] M. Odersky. The scala language specification, version 2.8, 2010. URL `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.

[14] World Wide Web Consortium (W3C). Html 4.01 specification, April 2014. URL `http://www.w3.org/TR/1999/REC-html401-19991224/`.