JadeIMP

A "slightly" improved version of Jade

By Saim Malik and Ahmed Khan

April 2016

1 Introduction

"Jade is a high performance template engine implemented with JavaScript." [1] Essentially it allows HTML content to be generated using all the basic programming principles (e.g. variable assignment, conditionals, loops, etc.). The syntax for Jade is also very similar to HTML with respect to how the file is structured, making it fairly easy to read. The main focus for this project was to recreate the current engine such that it will allow more dynamic changes to the HTML that gets generated. Simply put, the improved version of Jade - JadeIMP - allows manipulation of the generated HTML code through the client. This is accomplished by injecting JavaScript listeners into the generated HTML during compile time. Doing so allows variable values to be changed in real time, while simultaneously updating all the occurrences of that particular variable. This phenomenon is referred to as double binding; more on this will be discussed later.

1.1 Functionalities available with JadeIMP

- Generates pretty printed HTML code
- Allows basic programming features including variables, conditionals and loops
- Allows blocks of HTML code to be stored inside a variable (referred to as "block" in this document)
- Allows the generated HTML to be manipulated on the client side using "double binding"

2 Implementation

There are 6 phases of compilation. The six stages are lexical analysis, syntactic analysis, contextual analysis, intermediate code generation, code optimization and lastly code generation. The first three stages (lexical, syntactic, and contextual analyses) phases are called the analyzer, and the last three (intermediate code generation, code optimization, and code generation) are called the synthesis.

For the purposes of JadeIMP, we are converting JadeIMP code to HTML. For this our intermediate code is the same as our final target code, which is HTML. We also do not have a contextual analysis phase and code optimization phase. This was because the synthesizer did not need any additional information from the analyzer of the program. We have a file called index.nw which is the main program. This file will call the files synthesizer and analyzer to do their respective task.

This is a slight overview of what each file does in this project, to get additional information, go to the specified file.

2.1 analyzer.nw

This file is responsible for handling the analysis phase for the compiler. The analyzer file takes in the JadeIMP code and outputs an abstract syntax tree representation of it. It does this by calling the lexer.nw and parser.nw files respectively. This file was created to help isolate the analysis portion of the code with the synthesizer. This will help keep the code abstract and if we decided to change the structure of how either of the files work we would have more control to do so because the only function the synthesizer can access about the analyzer is the parse function and the convert function. For more information go to the analyzer.pdf file.

2.2 lexer.nw

This file does the lexical analysis. It will take in the JadeIMP code and will output a sequence of symbols that is used for the syntactic analysis. This file has one public function which is just the module that it exports. That function will do the corresponding task. For more information go to the lexer.pdf

2.3 parser.nw

This file does the syntactic analysis phase for the compiler. This file has one public function which is just the module that it exports. This function will take in a sequence of symbols that are token in from the lexer file and will create an abstract syntax tree from those symbols. For more information please go to the parser.pdf

2.3.1 Types of token

- Root: Outer most token that contains the contents of the entire file and all the variable information.

```
{
  "type":"root",
  "text":"<for debugging purposes>",
  "scope":
  {
```

```
    <object containing all the variables/blocks and their values;
        passed in by the client>
  },
  "content":[<list of tokens>]
}
```

- Directive: Token to keep track of all the different HTML tags (e.g. div, head, body, etc.), their attributes and their content (i.e. HTML that resides inside this directive).

```
{
  "type":"directive",
  "text":"<for debugging purposes>",
  "name":"<name of the directive>",
  "isClosing":<either true or false based on whether this directive
     should have a closing tag>,
  "attributes":[<list of attribute tokens>],
  "content":[<list of tokens>]
}
```

- Attribute: Token to keep track of all the attributes (for a certain directive); stores the name of the attribute (e.g. class, style, id, etc.) and the corresponding value.

```
{
  "type":"attribute",
  "text":"<for debugging purposes>",
  "parameter":"<name of the attribute>",
  "value":"<value of the attribute>"
}
```

- RawText: Token to keep track of textual content of the HTML page. The text could contain variables.

```
{
  "type":"rawText",
  "text":"<for debugging purposes>",
  "value":"<some text>"
}
```

- If: This token acts like an if-statement (a primitive tool in all programming languages). It keeps track of the condition of the if-statement, the if block (denoted by "statements" list) and else block. If the condition evaluates to true, the tokens in the statements list are parsed, otherwise the tokens in the else list are parsed.

```
{
  "type":"if",
  "text":"---",
  "condition":"<condition to execute if statement>",
  "statements":[<list of tokens>],
  "else":[<list of tokens>]
}
```

- For: Also a primitive tool in all programming languages. The for-loop token keeps track of the three statements normally inside a for-loop structure; the variable declaration (e.g. "var i = 0"), the condition at which the loop will stop, speed of the loop (e.g. "i++") and the content that will be parsed in each iteration of the loop.

```
{
  "type":"for",
  "text":"---",
  "declaration":"<variable declaration>",
  "condition":"<condition to stop the loop>",
  "iteration":"<statement executed after each iteration>",
  "content":[<list of tokens>]
}
```

- Block (in the scope object): The block token acts as a mini root token; such that it contains some content that can be referenced later in the file. It can be thought of as a variable that contains multiple tokens instead of just simple text.

```
{
  "type":"block",
  "text":"---",
  "name":"<name of block; used when referencing this block>",
  "content":[<list of tokens>]
}
```

- Block (for referencing): Token that essentially contains the name of the block that is being referenced.

```
{
  "type":"block",
  "text":"<for debugging purposes>",
  "name":"<name of block being reference>"
}
```

- Variable (in the scope object): The variable is stored as a simple key,value pair in the global scope object

```
{
  "<variable name>":<variable value>
}
```

2.4 synthesizer.nw

After the tree (JSON object) has been generated by the parser, it is the job of the synthesizer to convert the tree into HTML code. It requires two parameters, the tree and the scope object (to get access to all the variable and block values). Essentially, the code recursively goes down a path in the tree until it reaches the leaf node (usually a rawText token), retraces its steps and repeats the process until the entire tree has been traversed. Doing so outputs pretty printed HTML code that too has a tree like structure.

2.4.1 Double binding

The term "double binding" is mentioned in several places in this document. The synthesizer is where that functionality it handled. Whenever a variable reference is encountered (inside rawText token), besides replacing the referencing text by its value (retrieved from the scope object), the added value is surrounded by a "" tag with a "class" attribute which has a value unique to that variable's name. An example of this can be seen as follows:

(Note: it is assumed that the scope contains the variable "firstname":"Saim" for the following examples)

- Original text (as retrieved from the tree):
  My name is $firstname
- Text generated by the synthesizer:
  My name is Saim

A similar thing is done with directive tokens with "name":"input" (i.e. <input> tag in HTML). If the input directive contains an attribute called "bind", the variable in the attribute's value is assumed to be bound to the <input> tag and vice versa (hence the term double binding). Like before, this too is injected with a "class" attribute with a unique value. An example of this can be seen below:

- Original input directive (Note: The "isClosing":false indicates that the HTML will not contain a closing tag for this directive):

```
{
  "type":"directive",
  "text":"<for debugging purposes>",
  "name":"input",
  "isClosing":false,
  "attributes":[
    {
      "type":"attribute",
      "text":"<for debugging purposes>",
      "parameter":"bind",
      "value":"firstname"
    }
  ],
  "content":[]
}
```

Equivalent HTML of this would be: <input bind="firstname">

- Text generated by the synthesizer:
  <input class="variable_in_firstname" value="Saim">

So why inject the generated code with unique class names? The reason for this is b/c it makes it very easy for the client side script to manipulate these values. The library used to dynamically manipulate the generated HTML is called JQuery [3]. An example of an injected JQuery snipped can be seen below (still continuing from examples above):

```
// Surround with <script> tags to denote JavaScript code
<script type="text/javascript">
  // Set a listener that fires every time a change is made to an <input> with the given
       class name (i.e. "variable_in_firstname" - works for multiple <input>s that have been
        set to "bind" with the same variable)
  $(".variable_in_firstname").on("input", function() {
    // Temporarily remove the class so current <input> doesn't glitch
    $(this).removeClass("variable_in_firstname");
    // Get the changed value and apply it to all the references of the given variable (i.e.
         tags with class="variable_firstname") and also to all <input> (with class="
         variable_in_firstname")
    var var_val = $(this).val();
    $(".variable_firstname").text(var_val);
    $(".variable_in_firstname").val(var_val);
    // Add the class back so the current <input> is in the same state as before
    $(this).addClass("variable_in_firstname");
  });
</script>
```

Listing 1: Example double binding script

The process outlined above is done for each variable reference, hence putting the "IMP" in JadeIMP.

3 Testing

For this project we had two types of testers, called integration testers and unit testers. All the testing files were placed in a folder called test. Inside the test folder we have two folders called integration and unit tests for the appropriate testing.

3.1 Integration

The integration test has a file called test.js and 3 folders called html, JadeIMP, and tree. Inside each folder we have sample files. the JadeIMP folder has a set of files which have JadeIMP code in them. In the tree folder we have files that have the abstract syntax tree representation of the JadeIMP code. The html folder has the expected html output files from the given abstract syntax tree.

For example we have a file called basic_directive1.jimp inside the JadeIMP folder, basic_directive1.json in the tree folder, and basic_directive1.html in the html folder. We have a file called test.js. This javascript files takes all the files in the JadeIMP folder and runs the lexical analyzer on them to convert it to an abstract syntax tree. It will then compare the one formed from the code to the the one in the tree folder to check if it is the same. The next step will be to convert the abstract syntax tree to html by using the synthesizer. The test.js code will then compare the html that is generated from the synthesizer with the one inside the folder html and will throw an error if they are not the same.

3.2 Unit

There is a folder called unit inside the test folder, inside the folder we have noweb files with the extension nw. Those files are used as test files. Inside them we have a noweb tag called test. this allows us to properly test private functions without changing the structure of our code or making them public. In the makefile we have a variable called debug if it is equal to the string "test." It will compile the tag filename.test.nw, if the string is empty it will compile the tag filename.nw. This will allow the test cases to run when the debug flag is true, and will remove the test cases when they are false.

3.3 Testing libraries

For testing we used a library called chai, and the mocha framework. mocha is a framework for testing your code. It allows you to organize your test cases and gives you a clean output of how many test cases pass, fail and the error that caused the test case to fail.

Chai is an npm module that lets you handle test cases with ease. In chai, we decided to test our code with expect, For example to test if the type of the variable is a given type the code is expect(input).to.be.an("object"), This makes code easy to read, and understand. and when test cases fail it can help you clearly identify what is causing the problem.

4 How to use

4.1 Installation

To install JadeIMP, type the following in your terminal:

```
git clone "https://github.com/AhmedAKhan/JadeIMP"
cd JadeIMP
npm install
```

This will install JadeIMP in the current folder. It is suggested that you install it in the node_modules folder of the program.

4.2 Usage

After importing the module, the following command can be used to compile the JadeIMP code.

```
jadeimp.compile(source);

// source   { String } is the jadeimp code that you want to convert to
    html
// scope    { object } {optional} should have all the variables in the
    jade code that is defined
// returns  { function } to generate the html from an object containing
    scope
```

The render function then allows the user to generate the HTML.

```
jadeimp.render(source, scope)

// source   { String   } is the jadeimp code that you want to convert to
     html
// scope    { object   } {optional} should have all the variables in the
     jade code that is defined
// returns  { String } html code as a string
```

4.3 Example

Following is an example of the entire procedure.

4.3.1 Example.jadeimp

```
head
  title.
    Jade vs JadeIMP
```

```
h1.
  JadeIMP
div
  input(type="text" value="" bind="name")
p.
  Your name is $name
div.
  The HTML is generated on the server side (same as Jade). But injecting
       certain scripts into the generated code allows the HTML to be
      manipulated on the client side.
```

4.3.2 Example.js

```
var fs = require("fs");
var jadeimp = require("./JadeIMP/index");

var scope = { "name":"Saim" };
var fileContent = fs.readFileSync("./Example.jadeimp", "utf8");
var htmlResult = jadeimp.render(fileContent, scope);

fs.writeFileSync("./Example.html", htmlResult);
```

4.3.3 Example.html

```
<html>
<head>
    <title>
      Jade vs JadeIMP
    </title>
    <script src="https://code.jquery.com/jquery-2.2.2.min.js" integrity="
        sha256-36cp2Co+/62rEAAYHLmRCPIych47CvdM+uTBJwSzWjI=" crossorigin="
        anonymous"></script>
</head>
<body>
  <h1>
    JadeIMP
  </h1>
  <div>
    <input type="text" value="Saim" class="variable_in_name">
  </div>
  <p>
    Your name is <span class="variable_name">Saim</span>
```

```
  </p>
  <div>
    The HTML is generated on the server side (same as Jade). But
        injecting certain scripts into the generated code allows the HTML
        to be manipulated on the client side.
  </div>

  <!-- This script is automatically generated to allow double binding -->
  <script type="text/javascript">
    $(".variable_in_name").on("input", function() {
      $(this).removeClass("variable_in_name");
      var var_val = $(this).val();
      $(".variable_name").text(var_val);
      $(".variable_in_name").val(var_val);
      $(this).addClass("variable_in_name");
    });
  </script>
</body>
</html>
```

References

[1] PugJS. *Pugjs/pug, GitHub, 2016*. [*Online*].
Available: https://github.com/pugjs/pug. Accessed: Mar. 16, 2016.

[2] AngularJS. *2016, "AngularJS — Superheroic JavaScript MVW framework," 2008*. [*Online*].
Available: https://angularjs.org/. Accessed: Mar. 16, 2016.

[3] JQuery. *jq. Foundation, "JQuery," 2016*. [*Online*].
Available: https://jquery.com/. Accessed: Mar. 16, 2016.