WAEBRIC: a Little Language for Markup Generation

Tijs van der Storm

June 5, 2009

Abstract

WAEBRIC is a small language for generating XHTML markup. Its design is motivated by the lack of programmer friendly abstraction facilities in existing markup languages. WAEBRIC provides a user-friendly syntax to factor Web pages in self-contained functional building-blocks. This report introduces and motivates WAEBRIC, and presents its syntax and semantics.

1 Introduction

1.1 Motivation

- (X)HTML is too verbose to be typed (or read) by humans.
- Template languages feature elaborate quoting schemes that make designing templates form (X)HTML even more cumbersome.
- Template languages often do not allow functional abstraction and/or recursion.
- Template languages do not support "around" parameterization where one can reuse a piece of markup with one or more holes in it.
- Template languages often allow arbitrary computation thus violating separation of concerns guidelines.
- WYSIWYG editors have their own issues like generating inaccessible and unmaintainable XHTML code.

1.2 What WAEBRIC offers

WAEBRIC is a small programming language to factor web pages in reusable, function building blocks. Concretely, this amounts to the following:

- A WAEBRIC program consists of a number of function declarations. A function may accept a number of parameters and produces a piece of markup. Functions can be recursive.
- Markup is produced using (pre-defined) "function calls" corresponding to the tags that are part of XHTML 1.1. Keyword parameters to these functions correspond to XML attributes of the tag in question.
- The builtin operators **echo**, **comment**, **cdata** are used to produces text content, XML comments, and CDATA sections respectively.
- Limited control flow is provided through the **if**/**else** and the **each** iteration construct. Both these constructs operate on expressions (i.e. data, *not* markup).

- The special statement **yield** is provided to parameterize functions with additional markup. Markup arguments to a function invokation will be output where (one or more) **yield** is encountered. This mechanism is similar to (but weaker than) how Ruby block arguments are used in Markaby[1].
- WAEBRIC provides special syntax for common attributes in XHTML, such as "class" and "id". This is inspired by HAML[2].

In the design of WAEBRIC, explicit care is taken that *data* can be output as markup, but markup can *never* act as data. Thus it is not possible to "compute" with generated markup. Furthermore, to enforce strict model-view separation, computation with data is limited to testing the type or presence of data and looping through data using *each*.

2 Syntax

2.1 A Simple Example

A WAEBRIC file always starts with the keyword **module** followed by an identifier that should correspond to the basename of the file. A typical WAEBRIC program looks as follows:

```
1  module homepage
2
3  site
4    index.html: home("Hello World!")
5  end
6
7  def home(msg)
8    html {
9      head title msg;
10     body echo msg;
11   }
12 end
```

This program should be in a file called `homepage.wae`. This module contains one site definition that states that `index.html` should contain the output of evaluating the function `home` with a single, string-valued argument.

If we look at the `home` function, we see that it constructs a standard HTML document containing a header with a title element and a body. Both the contents of the title element and the body will be the value passed in as `msg`. Since `html` is not defined in this module and since it is not imported either, WAEBRIC assumes it is part of XHTML 1.1 and will output the corresponding tags. Within those tags, it will generate the output of the statements enclosed in curly braces.

The `html, head, title, body` elements in the example are called *markup* statements (or just *markup*). Markup can be nested by juxtaposing function calls and/or standard element names. For instance, the title element (containing msg) will be contained in the head element. The last item in a sequence of markups can be a statement or an expression. An example of the former is the block enclosed in curly braces which is passed into the html element. The nesting of msg in the title element is an example of the latter.

2.2 Embedding

2.3 Caveat

Some notes are in order with respect to how markup juxtaposition is parsed in WAEBRIC. Basically, it boils down to the following two observations. First, an single identifier interpolated in a string is

[1] `http://markaby.rubyforge.org/`
[2] `http://haml.hamptoncatlin.com/`

parsed as an expression whereas a single identifier in a statement context is parsed as markup (e.g. a function call or an XHTML element construction). Secondly, an identifier in the last position of a markup chain/spine is parsed as a variable (not as markup). The following examples illustrate these rules:

```
1  p; // markup
2  p p; // markup, variable
3  p p(); // markup, markup
4  echo "⟨p⟩"; // variable
5  echo "⟨p()⟩"; // markup
6  echo "⟨p p⟩"; // markup, variable
7  echo "⟨p p()⟩"; // markup, markup
```

As the example shows, parentheses can be used to force the parser to see an identifier as markup.

2.4 Data

It is not possible to compute with data in WAEBRIC, however, you can create and inspect data to a certain extent. WAEBRIC contains literal syntax for numbers, strings, symbols, lists and records (inspired by JSON[3]). Some examples are listed below:

```
1  123 // a number
2  "abc" // a string
3  'sym // a symbol
4  [123, "abc", 'sym] // a list
5  { name : "John Smith", age: 30 } // a record
```

The inspection of data is limited to testing whether an expression is of list or record type using x.list? and x.record? respectively, and obtaining a field of a record the using the dot-notation $x.y$. Testing the type of an expression can only occur in the condition of an if-statement. Finally, lists can be iterated through using the **each** statement.

2.5 Yield

In the example above, the html markup-invocation receives a block enclosed in curly braces that will be the content of the <html></html> element. Users can define functions that exhibit the same behaviour using yield. Consider the following refactoring of the example above:

```
1  def home(msg)
2    layout(msg) echo msg;
3  end
4
5  def layout(title)
6    html {
7      head title title;
8      body yield;
9    }
10 end
```

The home function now calls an auxiliar function (layout) which receives the statement **echo** msg as block-argument. The layout function inturn sets up a basic HTML skeleton (with the title argument as title) passes the **yield** statement to the body-tag. This has the effect that in the invocation of layout above, the output of **yield** will be equal to the output of **echo** msg.

Although the input-output behaviour of home is exactly the same as it was before, the page-skeleton as embodied in the layout function can now be reused in the generation other pages.

[3] http://www.json.org/

Shorthand	Equivalent longhand
div.x	div(class=x)
div#x	div(id=x)
input$$x$	input(name=x)
input:x	input(type=x)
img@w%h	img(width=w, height=h)

Table 1: Meaning of attribute shorthands

```
1  def menu(menu)
2    echo menu.title;
3    ul each (kid: menu.kids)
4        menu-item(kid);
5  end
6
7  def menu-item(item)
8    if (item.kids)
9      li menu(item);
10   else
11     li a(href=item.link) item.title;
12 end
```

Menu

- Home
- Misc
  - Contact
  - Links

Figure 1: Recursive menus in WAEBRIC and the possible output

2.6 Attributes

2.6.1 Introduction

Both function calls and markup elements can receive parameters. They come in two forms:

1. $f(x_1 = a_1, ..., x_n = a_n)$: if f is a tagname (for instance "div"), $x_1, ..., x_n$ are interpreted as XML attributes. The values are obtained from the values of the $a_1, ..., a_n$ respectively. In the case of a function calls, the keywords are ignored, and $a_1, ..., a_n$ is passed as an ordinary list of arguments (see 2 below).

2. $f(a_1, ..., a_n)$: if f is a defined function, $a_1, ..., a_n$ are its arguments. In the case of markup (for instance if f is actually the tagname "input"), the list $a_1, ..., a_n$ is translated to the the keyword parameter list value $= a_1, ...,$ value $= a_n$. In the case that $n > 1$ the value of "value" attribute is taken to be the value of the last keyword argument.

2.6.2 Shorthands

WAEBRIC provides shorthand notation for common XHTML attributes (inspired by Haml). See Table 1 for an overview of how this shorthand notation corresponds to regular attribute specification.

The syntax of shorthands accepts identifiers as variable part (except for the @-attribute which accepts natural numbers). The height part of the @-attribute is optional. Shorthands can be combined. For instance: p#intro.dropcaps combines the shorthands for the id and class attributes.

2.7 Recursive Menus

The example in Figure 1 shows how recursive menus could be defined in WAEBRIC. The first function, menu, receives a data object (*menu*) containing the labels, URLs and sub-menus that should be rendered in XHTML. The next statement just renders the title of the (current) menu using the built in state-

ment echo. After the title follows an unordered list containing the items of this menu. For each element in the "kids" property of *menu* the menu-item function is called.

The menu-item function first checks whether this *item* has any children (sub-menus). If so, it produces a li(st) element containing the output of a recursive call to menu. If there are no sub-menus, the result is a list element with an anchor tag which links the title of item to its URL. The result of an invocation (with the appropriate data for the *menu* parameter) of menu could look like the screen shot next to the source code.

3 Well-formedness

Apart from syntactic well-formedness, a WAEBRIC implementation also performs modest semantic well-formedness checks. The interpreter and compiler, however, will always produce output, regardless of errors. The following warning signs are checked for (the text between [] indicates the effect on evaluation).

Undefined functions If for a markup-call, f, no function definition can be found in the current module or one of its (transitive) imports, and, if f is not a tag defined in the XHTML 1.0 Transitional standard, then this is an error. [f will be interpreted *as if* it was part of XHTML 1.0 transitional.]

Undefined variables If a variable reference x cannot be traced back to an enclosing let-definition or function parameter, this is an error. [The variable x will be *undefined* and evaluate to the string "undef".]

Non-existing module If for an import directive `import` m no corresponding file m`.wae` can be found, this a an error. [The import directive is skipped]

Duplicate definitions Multiple function definitions with the same name are disallowed[4].

Arity mismatches If a function is called with an incorrect number of arguments (as follows from its definition), this is an error. [If a function is called with more actual arguments than the number of formal arguments, the superfluous arguments are ignored. If the actual arguments are too few in number, the remaining formal parameters become undefined variables.]

[4] Note that it is possible to shadow standard XHTML tags, for instance, by defining a function p.

Appendix: grammar in SDF

```
definition
module languages/waebric/syntax/Waebric

imports
      languages/waebric/syntax/Comments
      languages/waebric/syntax/Modules


hiddens
context-free start-symbols Module
module basic/IdentifierCon

exports

sorts IdCon

lexical syntax

head:[A-Za-z] tail:[A-Za-z\-0-9]* -> IdCon {cons("default")}

lexical restrictions

IdCon -/- [A-Za-z\-0-9]
module languages/waebric/syntax/Sites

imports languages/waebric/syntax/Markup

exports
sorts Mapping FileName DirName Site Path FileExt Directory PathElement

context-free syntax
  "site" {Mapping ";"}* "end" -> Site {cons("site")}
  Path ":" Markup -> Mapping {cons("mapping")}
  DirName "/" FileName -> Path {cons("path")}
  FileName -> Path
  Directory -> DirName

context-free restrictions
  DirName -/- ~[\/]
  "/" -/- [\ \t\n\r\.\/\\]

lexical syntax
  ~[\ \t\n\r\.\/\\]+ -> PathElement
  [a-zA-Z0-9]+ -> FileExt
  {PathElement "/"}+ -> Directory
  PathElement "." FileExt -> FileName

lexical restrictions
  FileExt -/- [a-zA-Z0-9]
module basic/NatCon
```

```
exports

sorts NatCon

lexical syntax

 [0-9]+  → NatCon {cons("digits")}

lexical restrictions

 NatCon -/- [0-9]
module languages/waebric/syntax/Modules

imports
      languages/waebric/syntax/Functions
      languages/waebric/syntax/Sites

exports

sorts ModuleId Import ModuleElement Module

context-free syntax
  {IdCon "."}+  → ModuleId {cons("module-id")}
  "import" ModuleId  → Import {cons("import")}

  FunctionDef  → ModuleElement
  Import  → ModuleElement
  Site  → ModuleElement
  "module" ModuleId ModuleElement*  → Module {cons("module")}

context-free syntax
  "module"  → IdCon {reject}
  "import"  → IdCon {reject}
  "def"  → IdCon {reject}
  "end"  → IdCon {reject}
  "site"  → IdCon {reject}
module basic/Whitespace

exports
  lexical syntax
    [\ \t\n\r]  → LAYOUT {cons("whitespace")}

  context-free restrictions
    LAYOUT? -/- [\ \t\n\r]
module languages/waebric/syntax/Statements

imports languages/waebric/syntax/Expressions
      languages/waebric/syntax/Predicates
      languages/waebric/syntax/Markup
      basic/StrCon

hiddens
context-free start-symbols
```

```
  Statement

exports
sorts Statement Assignment Formals

context-free syntax
  "if" "(" Predicate ")" Statement NoElseMayFollow  → Statement {cons("if")}
  "if" "(" Predicate ")" Statement "else" Statement  → Statement {cons("if-else")}
  "each" "(" IdCon ":" Expression ")" Statement  → Statement {cons("each")}
  "let" Assignment+ "in" Statement* "end"  → Statement {cons("let")}
  "{" Statement* "}"  → Statement {cons("block")}

context-free syntax
  "comment" StrCon ";"  → Statement {cons("comment")}
  "echo" Expression ";"  → Statement {cons("echo")}
  "echo" Embedding ";"  → Statement {cons("echo-embedding")}
  "cdata" Expression ";"  → Statement {cons("cdata")}
  "yield" ";"  → Statement {cons("yield")}

context-free priorities

context-free syntax
  IdCon "=" Expression ";"  → Assignment {cons("var-bind")}

context-free syntax
  "if"  → IdCon {reject}
  "comment"  → IdCon {reject}
  "echo"  → IdCon {reject}
  "cdata"  → IdCon {reject}
  "each"  → IdCon {reject}
  "let"  → IdCon {reject}
  "yield"  → IdCon {reject}

sorts NoElseMayFollow

context-free syntax
   → NoElseMayFollow

context-free restrictions
  NoElseMayFollow -/- [e].[l].[s].[e]

context-free syntax
  IdCon "(" {IdCon ","}* ")" "=" Statement  → Assignment {cons("func-bind")}
module languages/waebric/syntax/Functions

imports languages/waebric/syntax/Statements

exports

sorts FunctionDef Formals

context-free syntax
  "(" {IdCon ","}* ")"  → Formals {cons("formals")}
   → Formals {cons("empty")}
```

```
  "def" IdCon Formals Statement* "end"  → FunctionDef {cons("def")}
module basic/StrCon

exports

sorts StrCon StrChar

lexical syntax
   "\\n"  → StrChar {cons("newline")}
   "\\t"  → StrChar {cons("tab")}
   "\\\""  → StrChar {cons("quote")}
   "\\\\"  → StrChar {cons("backslash")}
   "\\" a:[0-9]b:[0-9]c:[0-9]  → StrChar {cons("decimal")}
   ~[\0-\31\n\t\"\\]  → StrChar {cons("normal")}

   [\"] chars:StrChar* [\"]  → StrCon {cons("default")}
module languages/waebric/syntax/Comments

exports

sorts Comment CommentChar Asterisk

lexical syntax
   "/*" CommentChar* "*/"  → Comment {category("Comment")}
   ~[\*]  → CommentChar
   Asterisk  → CommentChar
   Comment  → LAYOUT
   [\*]  → Asterisk
   "//" ~[\n]* [\n]  → Comment {category("Comment")}

lexical restrictions
   Asterisk -/- [\/]

context-free restrictions
   LAYOUT? -/- [\/].[\/]
   LAYOUT? -/- [\/].[\*]
module languages/waebric/syntax/Embedding

imports languages/waebric/syntax/Markup
      languages/waebric/syntax/Text

exports

sorts PreText PostText MidText TextTail Embed Embedding

lexical syntax
  "\"" TextChar* "<"  → PreText
  ">" TextChar* "\""  → PostText
  ">" TextChar* "<"  → MidText

context-free syntax
  PostText  → TextTail {cons("post")}
  MidText Embed TextTail  → TextTail {cons("mid")}
  PreText Embed TextTail  → Embedding {cons("pre")}
```

```
context-free syntax
  Markup* Expression  → Embed {cons("exp-embedding")}

context-free priorities
  Markup* Markup  → Embed {cons("markup-embedding")}
  >
  Designator  → Markup
module languages/waebric/syntax/Predicates

imports languages/waebric/syntax/Expressions

exports

sorts Predicate Type

context-free syntax
 "list"  → Type {cons("list-type")}
 "record"  → Type {cons("record-type")}
 "string"  → Type {cons("string-type")}

context-free syntax
  Expression  → Predicate
  Expression "." Type "?"  → Predicate {cons("is-a")}

context-free priorities
  "!" Predicate  → Predicate {cons("not")}
  >
  {left:
   Predicate "&&" Predicate  → Predicate {cons("and"), left}
   Predicate "||" Predicate  → Predicate {cons("or"), left}
  }
module languages/waebric/syntax/Text

hiddens
context-free start-symbols
  Text

exports

sorts Text TextChar EscQuote Amp TextCharRef TextEntityRef

lexical syntax
  "\"" TextChar* "\""  → Text
  ~[\0-\31\&\"\<\128-\255] \/ [\n\r\t]  → TextChar
  [\\][\"]  → EscQuote
  [\&]  → Amp
  Amp  → TextChar
  EscQuote  → TextChar
  TextCharRef  → TextChar {category("Constant")}
  TextEntityRef  → TextChar {category("Constant")}


  %% Copied from XML grammar
```

```
  "&#" [0-9]+ ";"  → TextCharRef
  "&#x" [0-9a-fA-F]+ ";"  → TextCharRef
  "&" [a-zA-Z\_\:] [a-zA-Z0-9\.\-\_\:]* ";"  → TextEntityRef


lexical restrictions
  Amp -/- [\#0-9a-zA-Z\_\:]
module languages/waebric/syntax/Markup

imports languages/waebric/syntax/Expressions
        languages/waebric/syntax/Statements
        languages/waebric/syntax/Embedding

hiddens
context-free start-symbols Markup

exports
sorts Markup Designator Attribute Arguments Argument

context-free syntax
  "(" {Argument ","}* ")"  → Arguments {cons("args")}
  IdCon "=" Expression  → Argument {cons("attr")}
  Expression  → Argument

context-free syntax
  Designator Arguments  → Markup {cons("call")}
  Designator  → Markup

context-free priorities
  Markup+ Statement  → Statement {cons("markup-stat"),non-assoc}
  >
  {
  Markup ";"  → Statement {cons("markup")}
  Markup+ Markup ";"  → Statement {cons("markup-markup")}
  Markup+ Expression ";"  → Statement {cons("markup-exp")}
  Markup+ Embedding ";"  → Statement {cons("markup-embedding")}
  }

context-free priorities
  Markup+ Markup ";"  → Statement
  >
  Designator  → Markup

context-free syntax
  IdCon Attribute*  → Designator {cons("tag")}

context-free syntax
  "#" IdCon  → Attribute {cons("id")}
  "." IdCon  → Attribute {cons("class")}
  "$" IdCon  → Attribute {cons("name")}
  ":" IdCon  → Attribute {cons("type")}
  "@" w:NatCon "%" h:NatCon  → Attribute {cons("width-height")}
  "@" w:NatCon  → Attribute {cons("height")}
module languages/waebric/syntax/Expressions
```

```
imports basic/Whitespace
        basic/NatCon
        basic/IdentifierCon
        languages/waebric/syntax/Text

exports
sorts SymbolCon SymbolChar Expression KeyValuePair

lexical syntax
  "'" SymbolChar* -> SymbolCon
  ~[\0-\31\)\ \t\n\r\;\,\>\127-\255] -> SymbolChar

lexical restrictions
  SymbolCon -/- ~[\)\ \t\n\r\;\,\>]

context-free syntax
  Text -> Expression {cons("text")}

context-free syntax
  IdCon -> Expression {cons("var"), category("MetaVariable")}
  SymbolCon -> Expression {cons("sym")}
  NatCon -> Expression {cons("num")}

context-free priorities
  Expression "." IdCon -> Expression {cons("field")}
  >
  Expression "+" Expression -> Expression {cons("cat"), left}


context-free syntax
  "[" {Expression ","}* "]" -> Expression {cons("list")}
  "{" {KeyValuePair ","}* "}" -> Expression {cons("record")}
  IdCon ":" Expression -> KeyValuePair {cons("pair")}
```