Composable Templates

Mario Blažević
Senior Software Developer
Stilo International
1900 City Park Drive
Suite 504
Ottawa
Ontario K1J 1A3
Canada

1 Introduction

A template language is a domain-specific programming language which is, syntactically, a superset of its output. In other words, any valid instance of the target output, with the minor exception of character escaping issues, is itself a valid template.

There are hundreds of template languages in existence. A cursory search of the Web will turn up dozens of stand-alone and library implementations. Most template languages, however, serve as a hidden part of another software system: they can be found in publishing systems, content management systems, word processors, macro and meta-programming engines, and many other places. A well-known but perhaps non-obvious example of an embedded template language interpreter is the function **printf** from the C standard library.

The characteristic feature of template languages, the similarity of their syntax to the target output, is also the main reason for their popularity. The process of creating a template is declarative. A template writer can concentrate on specifying the output itself, not the process by which the output will be created. This also means that templates can be created and maintained by target-language experts, who need not be programmers. Another cause for their popularity is that implementing a template language interpreter is relatively trivial, and usually more interesting than the rest of the work.

Syntactically, a template language is usually a mixed sequence of the target language literals, which get reproduced in the output with no modifications, and non-literal expressions which get evaluated in some way. The syntax and semantics of the non-literal expressions can vary from simple positional markers, as in the first argument of **printf**, to an unrestricted Turing-complete programming language as with [PHP]. Most template languages try to strike a balance between these two extremes. On one hand, increasing the expressive power of the language increases the target scope of the template language. On the other hand, with expressivity comes increasing complexity.

Another common goal in template language design is the support and enforcement of *separation of concerns* [Parr 2004] between the template authors and software developers. Ideally, the templates should be concerned only with presentation, while the host software using the templates should feed them only the raw data with no formatting.

A template language can be, and usually is, restricted in two ways. First, the language control structures can be limited to the point that they cannot express any business logic that doesn't belong in the presentation layer. Secondly, the language I/O constructs can be restricted so that a template only has access to the input data necessary to produce its output: for example, no file access would be allowed. With this restriction in place, we can view a template as a pure function from the restricted inputs given by the host environment to its target output.

1-1 Filters

One way to increase the expressivity of a template language is by adding more control constructs to it. The first to be added are usually loops and conditionals, then variables, even arithmetic, until the language becomes Turing-complete. Alternatively, we can make the host software perform the necessary computations on demand and supply the results to the template. In other words, we can provide the templates with a function library. This will be our preferred route.

One class of functions that is often made available in template languages is *text filters*. Filters solve a practical problem familiar to anybody who has generated HTML, XML, or SGML output: all special characters in the textual input must be escaped before they can be output as data content [1]. The algorithm necessary to perform this escaping, however, cannot be expressed in a template language without some rather sophisticated extensions. Many popular template engines like **[Django]** therefore choose to provide text filters like **escape**, **center**, and **pluralize** ready-made.

While the addition of text filters to a template language obviously makes the language more difficult to learn, if the set of predefined filters is chosen carefully it can be used intuitively by non-programmers. The impact on separation of concerns, again, entirely depends on the filters that are made available to the template language. If a filter performs a high-level task that is usable only for presentation purposes, then the templates cannot violate the principle of separation of concerns by using it.

Another thing to examine is the impact of text filters on the I/O allowed to templates. If we have chosen to restrict the inputs available to the templates, the easiest and safest way to enforce this restriction in the presence of filters is to make all filters pure functions. In other words, a text filter can use only the input provided to it by the template and cannot have any effects other than feeding its result back to the template.

1-2 Templates as Functional Programs

At this point, we have reached the conclusion that a template should be a pure function that generates its output value from a restricted set of input values. To be able to achieve this task, the template should have access to a set of predefined text filters, and these filters should also be pure functions.

It should not come as surprise at this point that, according to most definitions, our abstract template language is a *functional programming language*. If we accept this conclusion, what can this new perspective on template languages teach us?

Most functional programming languages encourage recursion. While conceptually simple, unrestricted recursion makes the language Turing-complete. It can also be difficult to understand. For the latter reason, modern functional languages supply a number of higher-order functions that can be used to express common patterns of recursion in a more controlled way. Programmers are encouraged to use the standard higher-order functions instead of writing their own recursive algorithms **[Backus 1978]**. Foremost examples among these standardized functions are *compose* and *map* (also referred to as *lift*). The former higher-order function composes two argument functions into a new function, and the latter applies its argument function to all members of a collection.

From our perspective, the interesting thing about *map* is that it acts like a loop construct, in that it invokes its function argument multiple times, once for every member of the collection argument. Unlike many loop constructs in general-purpose programming languages, however, it is very simple to understand and it provides a number of important guarantees: for example, it cannot be used to construct an infinite loop. We can say it is a *data driven* control structure, because its behaviour is driven by the shape of its collection argument.

1-3 An example

To illustrate how control constructs can be replaced by higher-order functions, we shall use a fragment of an example taken from the documentation of Django **[Django]** template language:

```
{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
```

The template fragment above is a pure function from **story_list** to HTML. The **story_list** input is a list of records containing fields **get_absolute_url**, **headline**, and **tease**. The example uses a control construct, namely the **for** loop, text filters **upper** and **truncatewords**, as well as field accessors **get_absolute_url**, **headline**, and **tease**. From a functional programmer's point of view, all of the above are pure functions:

- Each of the field accessors is a function from a story to one of its fields.
- Text filters are functions from text to text.
- The vertical bar symbol **|** is a higher-order function commonly known as *compose*, taking as arguments the two functions specified on its sides and returning their composition.
- The dot symbol **.** denotes application of the function on its right-hand side to the value on its left.
- The body of the **for** construct is a template function that concatenates the literal elements like

```
<h2>
  <a href="
```

together with the results of function applications like

```
{{ story.get_absolute_url }}
```

This function is not one of the standardized higher-order functions, but it can be trivially defined. In Haskell **[Haskell 2003]**, for example, one definition would be as follows:

```
concatMapApply :: [a -> [b]] -> a -> [b]
concatMapApply functions input = concatMap ($ input) functions
```

- Finally, the **for** construct is an equivalent of the higher-order function *map*. The body of the construct is a function that gets applied to each item in the input list, and the result of the entire construct is the list of the function's results. If we don't care about the boundaries between different items of the result list, we can use *concatMap* instead of regular *map*. The former function concatenates together all the results, erasing the boundaries between them.

If we put together all the pieces, the example template could be written in Haskell as follows:

```
module Example where

import Char (toUpper)

concatMapApply :: [a -> [b]] -> a -> [b]
concatMapApply functions input = concatMap ($ input) functions

data Story = Story {get_absolute_url :: String,
                   headline :: String,
                   tease :: String}

upper = map toUpper
truncatewords max text = unwords (take max (words text))

template :: [Story] -> String
template = concatMap (concatMapApply [const "  <h2>\n",
                                      const "    <a href=\"",
                                      get_absolute_url,
                                      const "\">\n      ",
                                      upper . headline,
                                      const "\n",
                                      const "    </a>\n",
                                      const "  </h2>\n",
                                      const "  <p>",
                                      truncatewords 100 . tease,
                                      const "</p>\n"])


test = [Story "http://hello.org/" "Greeting" "This is a short greeting.",
        Story "http://hi.org/" "Another Greeting" "This is an even shorter greeting."]

main = putStrLn (template test)
```

Haskell is obviously not a template language. While the Haskell code above demonstrates how a template can be represented using higher-order functions, it is not a template itself. If we translate the code back into a template language, keeping its functional structure along the way, here is what it could look like:

```
[- #map ["
<h2>
  <a href="[- @get_absolute_url -]">
    [- @headline:upper -]
  </a>
</h2>
<p>[- @tease:truncatewords(100) -]</p>
"] -]
```

As this short example hopefully demonstrates, it is possible to design the semantics of a template language on the principles of higher-order functional programming without sacrificing the directness of its syntax. The actual template language used in this example will be presented in section 3 of the paper.

1-4 Data Model

The constructs the language provides for fetching input impose a certain view of the external world, both upon the template language and the template writer. This view is the *input data model*, and it represents one of the most distinguishing features of a template language.

The most restrictive kind of template languages provide a fixed number of input variables to each template. The input model in this case is a vector of input values. The template can specify the values it needs by their position, as in the **printf** example, or by name. Many template languages begin their life in this form.

The next addition to the data model tends to come in the form of multi-valued inputs. The reason for this extension is purely practical: many target outputs contain item lists, tables, etc. In order to be able to traverse the multiple values provided by the new input data model, the template language is usually extended by a looping construct. The looping construct is often semantically equivalent to the higher-order function *map*.

The logical next step is to add *nested* data collections, *i.e.*, lists of lists or S-expressions, to the input data model. With this extension in place, the templates become capable of generating outputs containing two-dimensional tables and nested lists.

Even as the *input* data model of a template language becomes more structured and expressive, the *output* data model is typically left as the same unstructured singleton value. The problem this dichotomy presents for templates as functions is that they cannot be composed: the output of one template cannot be fed into another template.

The solution to this impedance mismatch is to make template input and output have the same structure. One obvious candidate for this common structure is XML markup. Many of the existing template languages are used to generate markup: HTML, SGML or XML. Furthermore, XML can trivially encode nested data collections needed for the input data model of the template language.

The effect of this unification of input and output data model is to make every template a user-defined markup filter. These filters can be composed, mapped over components of XML documents, and combined in other ways using higher-order functions, to produce other markup filters. The goal of a template writer, then, is to construct a markup filter that will produce the target output from the input data encoded as an XML document.

1-5 Overview

The remainder of this paper will introduce OMDE [OmniMark Documentation Environment] template language, which tries to apply the above principles of template language design in practice. The next section will introduce the language and describe the context it's been developed in. The sections 3 and 4 define the language in detail, and section 5 explains its practical use through several examples. The results are presented in section 6, and the remaining sections 7 and 8 contain discussion of related work and the conclusion.

2 Context

2-1 History

OMDE stands for OmniMark Documentation Environment. It was originally developed in 1998. The OmniMark documentation before OMDE was a monolithic SGML document, and it was proving increasingly difficult to maintain. OmniMark [OmniMark 2007] is a programming language targeting text and markup processing.

The original OMDE developed in 1998 was based on the *microdocument architecture* [Baker 1998]. Documentation has been divided into many small, interlinked topics. Each topic could have multiple SGML documents associated with it, as well as links to other topics. Both the SGML documents and links were categorized according to their purpose. For example, most language keyword topics had associated *syntax* and *purpose* documents, and links could be categorized into *related keywords* and *related concepts*. Both the SGML content and the inter-topic links were stored in a database.

The goals of the OMDE project included simple authoring, content reuse, and multiple outputs. In order to achieve the first goal, the SGML markup for topic content was made minimal. For example, there are only two tags, **code** and **codeblock**, available for marking up code fragments. It was the task of the publishing engine to scan the code for keywords and link them to corresponding keyword topics. Content reuse was to be accomplished by inclusion of topic content into larger, *narrative* topics. Due to lack of demand, however, this feature did not see much use, and only one narrative document has ever been designed. As for the multiple outputs, originally there were three: HTML, Windows help, and PDF generated through FrameMaker.

Over the years, several problems were identified with OMDE:

- FrameMaker, used for generating the PDF and Windows Help documentation, was a constant bottleneck in the release process due to certain operations that had to be performed manually.
- The generated PDF was deemed to have poor quality.
- Distribution of the Windows Help format has later been abandoned in favour of HTML.
- The database that was used for the documentation had only a rudimentary support for versioning.
- The content authoring software, necessary because the documentation was stored in a proprietary database, was implemented in Visual Basic and worked only on Windows.

In the context of the present paper, the most significant problem was that the template language used by the OMDE publishing process was too simple: its data model allowed for singleton values only. As a consequence, much of the output formatting had to be specified by the host software, an OmniMark program, not by the templates. In that situation, only the initiated were (was?) able to modify the details of the generated output.

2-2 Data Migration

In December of 2006, it was decided that the OMDE processing software should be rewritten from scratch. The existing content would be reused with as little modification as possible. The proprietary database would be abandoned in favour of the combination of filesystem storage and Perforce SCM [Software Configuration Management] system for version control. The only output format produced in the beginning would be HTML, with PDF to follow later.

There are currently 3750 topics in OMDE, 2153 of them containing non-trivial SGML content. The published output consists of 1892 HTML files. It was decided that the only sane way to verifiably replicate the required functionality of the old OMDE was by producing the exact same output with all the existing bugs. Fixing the output issues could be left for a later day.

Microdocument architecture, proven as the best feature of the 1998 OMDE design, has been kept. Each topic, previously stored as a database record, became an SGML file instead. Inter-topic links, metadata, and SGML content moved from the record fields into the top-level elements of the topic file. The primary identifiers of topic records became file names, and database tables previously containing topics were mapped into directories containing topic files.

2-3 Processing Model

One of the project goals was to stress-test the new streaming capabilities of OmniMark 8. For this reason, processing logic has been organized into pipelines. Data originates from SGML topics, streams through various templates, and ends up as a collection of HTML files. The data stream is marked up in each phase of the process until the very end, when the markup is stripped from the stream before it gets written as plain text into the appropriate destination file.

Each template acts as a filter on a markup stream. There are top-level templates, largely matching the templates inherited from 1998, that closely mimic the output HTML pages. There are also templates that specify how a particular topic element, like paragraph, maps into HTML. Some templates contain no fragment of the output; they are used only to organize the input stream, usually by marking it up, before it can be processed into HTML.

It is arguable whether the templates of the latter kind really belong in the presentation layer. One example of a template that has nothing to do with the output format is the choice of the sorting and grouping criteria. In our opinion, this is more a matter of presentation than of business logic.

It must be emphasized that the word "pipeline", used here for lack of better terminology, should not be taken to imply linear processing. As already noted, the processing of elements from the SGML topic markup can be delegated to a smaller template filter, so the processing pipeline involves certain branching and division of tasks. Duplication and suppression of parts of the input stream also occur.

3 OMDE Template Language Definition

3-1 Data model

Some technical details aside, both input and output of a template are always markup streams. The markup model used in OMDE is much simpler than the full SGML/XML markup. Here is the whole of the model:

Markup Stream Data Model

Term	Definition
markup stream	sequence of *markup item*s
markup item	*character* or *element*
element	triple of *element name*, *element attributes*, and *element content*
element name	*string*
element attributes	map from *attribute name*s to *attribute value*s
attribute name	*string*
attribute value	*markup stream*
element content	*markup stream*
string	sequence of *character*s

Compared to SGML and XML, the markup model defined above is missing processing instructions, entities, comments, and marked sections. For the sake of simplicity, the model also removes some restrictions present in

SGML and XML: there is no concept of a single document element, and attribute values can be arbitrary markup streams. Element content is different from an attribute value only in that it is present in every element, though it may be empty.

The initial input to the template pipeline typically consists of a single *topic element* representing the topic to be published. Topic elements can also be produced within the template language in two ways: by accessing attributes of the predefined variable **$topics** and by using built-in filters **#find-topic-id** and **#match-topic-id**.

Every topic has certain metadata associated with it, as well as links to other topics. Both metadata and links are represented as attribute values of the topic element representing the topic. All topic metadata fields named **Index**, for example, are mapped to a single attribute with name **Index** and value equal to the sequence of all metadata field values individually wrapped in an element named **Index**. All links are mapped in the same way, except in this case the attribute value is the sequence of the topic elements representing the topics that the links are pointing to.

Aside from metadata and link attributes, every topic element has three predefined attributes: **topic-content**, **topic-id** and **topic-type**. The latter two contain the unique topic identifier and the topic type name.

For example, the template

```
[-$topics:@Library:#map[-@Name-]-]
```

would produce a list of all library names, and

```
[-$topics:@Library:#map["<a href="[-@topic-id-]">[-@Name:#escape-html-]</a>"]-]
```

would present all library names as HTML links.

The value of the attribute **topic-content** of a topic element represents the parsed SGML content of the topic. This value is generated on demand, which means that full parsing of the topic is delayed until necessary. As the topic content is parsed, the SGML elements within are mapped into the elements of the data model in a straightforward fashion. The remaining SGML markup is processed in the usual way: entities get expanded, processing instructions and comments ignored.

3-2 Template Syntax

The grammar of OMDE templates in EBNF form follows below.

OMDE Template Syntax

Non-Terminal	Definition
template	{*template item*}
template item	*character* | *filter composition*
filter composition	"**[-**" *filter* {"**:**" *filter*} "**-]**"
filter	*filter reference* | *template reference* | *built-in filter* [*filter argument*] | *variable reference* | *inline template* | *attribute reference* | *element literal*
filter reference	*name*

template reference	*name*
built-in filter	"**#**" *name* ["**(**" {*character*} "**)**"]
filter argument	*filter composition* | *inline template*
inline template	'**["**' *template* '**"]**'
variable reference	"**$**" *name*
attribute reference	"**@**" *name*
element literal	"**<**" *name* {*element attribute*} "**>**" *filter argument*
element attribute	*name* "**=**" *filter argument*
name	*letter* {*letter* | "**-**" | "_"}

All whitespace is significant in a *template*, and insignificant inside a *filter composition*. There are no comments built into the language, but they can be simulated by using unreachable inline templates.

The syntax presented above is ambiguous: an unqualified name can be a *filter reference* or a *template reference*. This ambiguity is resolved by looking up the name in question. New filters and templates can be added by creating a file or subdirectory in a special directory reserved for the purpose. The files specifying filters have the extension **.path** and their syntax matches *filter composition*, whereas the files specifying templates end with **.tpl** and contain the *template* syntax.

3-3 Template Semantics

The simplest template contains only text and no filter compositions. When such a template gets applied to a markup stream, the result is the template text.

In general, the result of evaluating a template on an input stream is the concatenation of results of evaluation of individual template components on the stream. If we denote the evaluation function as $\boldsymbol{E_{template}}$(*template*, *input*, *environment*), we can write the semantics more formally as follows:

$$
\begin{aligned}
E_{\text{template}}(T_1\,T_2 \ldots T_n,\, S,\, \text{env}) &= E_{\text{item}}(T_1,\, S,\, \text{env})\;\; E_{\text{item}}(T_2,\, S,\, \text{env}) \ldots E_{\text{item}}(T_n,\, S,\, \text{env}) \\
E_{\text{item}}(\text{Character}(c),\, S,\, \text{env}) &= c \\
E_{\text{item}}(\textbf{[}\text{-composition-}\textbf{]},\, S,\, \text{env}) &= E_{\text{composition}}(\text{composition},\, S,\, \text{env}) \\
E_{\text{composition}}(\text{filter}_1 \textbf{ : } \text{filter}_2 \textbf{ :} \ldots \textbf{:}\, \text{filter}_n,\, S_0,\, \text{env}_0) &= S_n \\
&\quad \text{where } \; (S_i,\, \text{env}_i) = E_{\text{filter}}(\text{filter}_i,\, S_{i-1},\, \text{env}_{i-1}) \\
E_{\text{filter}}(\textbf{[}\text{"template"}\textbf{]},\, S,\, \text{env}) &= (E_{\text{template}}(\text{template},\, S,\, \text{env}),\, \text{env}) \\
E_{\text{filter}}(\text{name},\, S,\, \text{env}) &= (E_{\text{composition}}(\text{ResolveFilter}(\text{name}),\, S,\, \text{env}),\, \text{env}) \\
&\quad \text{where } \; \text{name is a user-defined filter name} \\
E_{\text{filter}}(\text{name},\, S_1 S_2 \ldots S_n,\, \text{env}) &= (E_{\text{template}}(T_1,\, S_1,\, \text{env}) \ldots E_{\text{template}}(T_n,\, S_n,\, \text{env}),\, \text{env}) \\
&\quad \text{where } \; T_i = \text{ResolveTemplate}(\text{name},\, S_i) \\
&\quad \text{and } \; \text{name is a user-defined template name} \\
E_{\text{filter}}(\textbf{\$}\text{name},\, S,\, \text{env}) &= (\text{env}(\text{name}),\, \text{env}) \\
E_{\text{filter}}(\textbf{@}\text{name},\, \text{Element}(\text{element-name},\, \text{attributes},\, \text{content}),\, \text{env}) &= (\text{attributes}(\text{name}),\, \text{env}) \\
E_{\text{filter}}(\,\textbf{<}\text{name}\;\; \text{attr}_1 = V_1 \ldots \text{attr}_n = V_n \textbf{>}\; \text{content},\, S,\, \text{env}) &= \Big(\text{Element}\,(\text{name}, \\
&\qquad (\text{attr}_1 \mapsto E_{\text{arg}}(V_1,\, S,\, \text{env}),\, \ldots,\, \text{attr}_n \mapsto E_{\text{arg}}(V_n,\, S,\, \text{env})), \\
&\qquad E_{\text{arg}}(\text{content},\, S,\, \text{env})), \\
&\quad \text{env}\Big) \\
E_{\text{arg}}(\textbf{[}\text{-composition-}\textbf{]},\, S,\, \text{env}) &= E_{\text{composition}}(\text{composition},\, S,\, \text{env}) \\
E_{\text{arg}}(\textbf{[}\text{"template"}\textbf{]},\, S,\, \text{env}) &= E_{\text{template}}(\text{template},\, S,\, \text{env})
\end{aligned}
$$

3-3-1 Environment variables

When a filter composition is applied to an input markup stream, both the stream and the current environment first get processed by the leftmost filter in the composition, then the results get filtered through the second filter in the composition, *etc*, until the rightmost filter in the composition produces the output stream. The environment produced by the last filter is discarded; any modifications made to the environment (*i.e.*, variable assignments) are visible only within the innermost enclosing filter composition. Note that the scope of variable assignments is dynamic.

The initial environment contains only two variables: **$date**, containing the current date and time, and **$topics**, which contains the root of the topic tree represented as a single element with empty content. The element's attributes map each topic type name to the sequence of all topics of that type. The only way to modify the environment is by using the built-in filter **#let**.

3-3-2 User-defined filters and templates

The semantics of a user-defined name depends on whether the name represents a *filter reference* or a *template reference*, as described in section 3-2. In the former case, the referenced filter gets applied to entire input stream. If the name does not represent a filter reference, the input stream is split into individual items, and every item is processed separately. The template chosen for processing a particular markup item depends both on the given template name and on the markup item type. As a result, the same template reference may be resolved to different templates during the processing of an input stream with many items.

4 Predefined Filters

This section will present the list of all predefined filters in OMDE. Note that the choice of filters was driven by practical concerns. There are many filters that would make perfect sense among these, like **#last** or **#length**, but they were not added simply because no template has needed them so far.

One class of built-in filters operates on purely textual input with no elements. This kind of filters is quite common in Web template languages.

Text Filters

Syntax	Description
#ascii-code	Converts each character in the input text into its decimal representation of the character's ASCII code.
#characters	Wraps each character in the input text into an element named **Character**, with no attributes.
#escape-html	Escapes the characters with special meanings in HTML from the input: angle brackets, ampersand, and quotes.
#escape-html-content	Escapes the characters with special meanings in HTML content from the input: angle brackets and ampersand.
#escape-uri	Escapes the characters with special meanings in URIs from the input.
#find-topic-id(*topic type***)**	Scans the input and wraps each substring matching a valid *topic id* of the given *topic type* found in the input into the appropriate topic element.
#match-topic-id(*topic type***)**	Checks if its entire input matches a valid *topic id* of the given *topic type*. If so, wraps the input into the appropriate topic element, and otherwise drops it.
#lexicographic	Transform each textual item in the input into its canonical lexicographic form.
#lowercase	Converts all letters in the input text into lowercase.
#right-aligned (*W***)**	Aligns the entire input text to the right with the given constant width *W*.
#substring *S*	Evaluates *S* on the input text into *S'*, then outputs each occurrence of *S'* in the input text.
#trim-whitespace	Removes the leading and trailing whitespace from each top-level textual item.

The following filters expect a sequence of elements as input, and throw an error if they encounter any text outside an element content. Apart form this restriction, they are completely generic: with the exception of **#distribute**, the filters accept any sequence of well-formed elements. This generality is possible because all the filters except **#distribute** and **#content** are higher-level functions. They expect another function as argument, and any processing that is specific to the input can be delegated to this argument.

Element Filters

Syntax	Description
#content	Replaces each top-level element in the input stream with its content.
#copy *X*?	Replaces each element <a>*C* in the input with <a> $E_{arg(X,}$ <a>C) , the element wrapped around the result of evaluation of the given argument *X* on the element. If the argument is not specified, copies the entire element with its original content.
#distribute	Expects input with two levels of elements and no text. Each top-level element with *N* children gets replaced by *N* copies of the top-level element with one child each. For example, it would turn <a><b/><c/> into <a><b/><a><c/>.
#group-by *K*	For each contiguous sequence of elements $S = E_1 \ldots E_N$ where $E_{arg(K, E_i)}$ evaluates to the same result *R* for every element of the sequence, output a single element named **Group**, with a single attribute named **GroupKey** and valued *R*, and with the content *S*.
#if *T*	For each element *A* in the input, evaluates $E_{arg(T, A)}$. If the result is non-empty, outputs the original element *A*. If there is no output, suppresses the element.
#interpret-path *P*	Evaluates the given argument on each input element into *P'*, then outputs the result of evaluation of *P'* on the element.
#map *F*	Evaluates F on each element in the input and outputs all the results. Note that **#map** can be expressed as `#map[-F-] = #copy[-F-]:#content`
#no *T*	Opposite from **#if**. For each element *A* in the input, evaluates $E_{arg(T, A)}$. If the evaluation produces any output whatsoever, suppresses the element. If the result is empty, outputs the original element *A*.
#mark-duplicates *K*	Evaluates the key *K* on each input element. Whenever the result is the same for two consecutive elements, both are individually enclosed into an element named **Duplicate** with no attributes. Other elements are output unmodified.
#merge-sorted-groups *G*	Expects two streams of group elements, sorted by the value of **GroupKey** attribute: one in its input and the other as the result of evaluation of *G* on the input. Merges the two streams into a single sorted stream of groups.

#sort-by *K*

Evaluates $E_{arg}(K, A)$ on each input element *A*, then sorts the elements by the ascending value of the results.

The remaining built-in filters can handle mixed content as their input.

Mixed Content Filters

Syntax	Description
#apply *F*	If the input is not empty, applies F to the entire input. If the input is empty, the output of **#apply** is empty as well.
#drop-last	Reproduces the input except for the last character or element, which is dropped.
#first	Outputs only the first character or element from the input.
#flat-content	Eliminates all element tags from the input.
#id	Replicates the input.
#let *var* **=** *V*	Evaluates *V* on the entire input, stores the result value into variable *var* in the current environment, and replicates its entire input into the output.
#lines	Wraps each line of text in the input into an element named **Line** with no attributes. Top-level elements are treated the same as non-breaking characters.
#log	Passes the input through unmodified, logging its XML representation on the standard error stream at the same time.
#match *P*	Evaluates *P* on the entire input. If the input equals the result, it gets output through, otherwise it is suppressed.

5 Examples

5-1 Simple Lists

Here is a section of the template file **ToHTML/Library-topic.tpl**:

```
<td bgcolor="#e3e3e3" align=left valign=top>
<font size=2>
<b>Functions</b><br>
[-
   #let $TheLibrary=[-#copy-]:
   $topics:@Function:
   #if[-@FunctionLibrary:#match[-$TheLibrary-]-]:
```

```
      #sort-by[-@Name:#content:#lexicographic-]:
      #map
      ["   <a href="[-
            <TopicFileReference>[-@topic-id-]
         -]">[-
            @Name:#content:#escape-lessthan-in-html-content
         -]</a><br>[-nl-]"]
   -]
    
   </td>
   <td bgcolor="#e3e3e3" align=left valign=top>
   <font size=2>
   <b>Platforms</b><br>
   [-
      @Platform:#sort-by[-@Name:#content:#lexicographic-]:
      #map["   [-@Name:#content:#escape-html-content-]<br>[-nl-]"]
   -]
    
   </td>
```

The above template code expects a single **Library-topic** element in its input, and from this input it generates a list of all library functions and all platforms that the library works on. The main difference between the two lists is that the list of platforms can be obtained from the attribute **platforms** of the library topic, whereas the links between libraries and functions run in the opposite direction: every function topic has an attribute **FunctionLibrary** pointing to the appropriate library topic, but not the other way around. This organization of links makes the documentation maintenance easier: when a new function is added to a library, the library topic can be left unmodified.

The list of platforms is relatively easy to generate. The template reference **[-nl-]** inserts a newline. Inline templates can include literal newline characters and span multiple lines, but explicit naming of newlines like this can make them easier to spot.

```
@Platform:#sort-by[-@Name:#content:#lexicographic-]
```

grabs the list of all platforms, then sorts them lexicographically by the platform names. The following line maps an inline template over the list of platforms. The inline template formats each platform into HTML.

The function list generation is less straightforward because of the difference in linking. In order to get the list, we start with a list of *all* function topics:

```
$topics:@Function:
```

Then we filter the list to retain only those functions whose **FunctionLibrary** attribute links to our library:

```
#if[-@FunctionLibrary:#match[-$TheLibrary-]-]:
```

The variable *TheLibrary* has been previously set to the input library by the built-in filter **#let**. This assignment has no effect outside the composition. Once we obtain the list of functions, the rest of the pipeline is similar to the one for the platform list. The only difference is that we have to insert an HTML hyperlink to the page representing the function topic. We abstract from the exact URI of the target page by wrapping it into a **TopicFileReference** element. This element will be post-processed in the final stage of the processing pipeline, where the topics get written as HTML files and the topic references resolved into the relative URI paths.

5-2 Grouped Lists

The template file **generic/Keyword/Alpha-ToHTML.tpl** generates the alphabetical list of all keyword topics, grouped by first character of the keyword. The template contains the following lines that do the bulk of the work:

```
[-
   $topics:@Keyword:#copy[-@Entry-]:#distribute:
   #sort-by[-#flat-content:#lexicographic-]:
   #group-by[-#flat-content:#lexicographic:#first-]:FillGroupsAtoZ:
   #let $Line=[-TopicListToHTMLLine-]:#if[-#content-]:ListItemGroupToHTML
-]
```

This template expects input containing a single **Keyword-topic** element. Because a single keyword topic can describe several related OmniMark language keywords, every keyword topic has the potentially multi-valued attribute **Entry** with all the language keywords. These entries appear separately in the output list, but they all link to the same keyword topic.

Because there are many keywords in OmniMark, their alphabetical list is divided into section according to the leading character of the keyword. Every keyword section is preceded by a line containing all section links. The list is always the same, except the leading character of the current section must be emphasized and must not be a link. A further complication is that letters A-Z must all be present in the list; if there is no keyword starting with a particular letter, the letter should not be a link and must be made grey.

The first line of the above pipeline,

```
[-
   $topics:@Keyword:#copy[-@Entry-]:#distribute:
-]
```

prepares the list of all keyword topics, then fills the content of each keyword topic by all keyword entries it describes, and finally distributes the top-level **Keyword-topic** tags so they wrap individual entries. The next line first sorts the list of wrapped entries in lexicographic order, then groups the sorted entries by their first character, and fills in the missing letters. The latter task is accomplished by invoking the shared pipeline **generic/FillGroupsAtoZ.path** with the following contents:

```
#merge-sorted-groups[-
   AtoZ:#characters:#map["[-<Group GroupKey=[-#content-]>[""]-]"]
-]
```

The content of the referenced template file **generic/AtoZ.tpl** is simply

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The prepared list of sorted and grouped keyword entries is then used to set variable *Line* to a marked-up representation of the line containing the section links. The value of the variable is prepared by template file **TopicListToHTMLLine/Group.tpl**, whose content is:

```
[-
   #no[-#content-]:#apply["<font color="#999999">[-@GroupKey-]</font>"]
-][-
   #if[-#content-]:#copy[""]
-]
```

Note that the above template discards entire content of all character groups. If a group has no entries inside it, its content will be a grey character. If the group was not empty, its content gets removed completely.

With the section link line ready, the pipeline invokes the template in file **ListItemGroupToHTML/Group.tpl**:

```
<a name="[-@GroupKey:#lowercase-]"><font color="#9966CC">
[-#let $CurrentCharacter=[-@GroupKey-]:$Line:LineToHTML-]</font>
</a>
<br>
[-#content:BrokenListItemToHTML-]<br>
<br>
<hr>
```

The template gets invoked on a single **Group** element containing all entries that start with the same character. The line

```
[-#let $CurrentCharacter=[-@GroupKey-]:$Line:LineToHTML-]</font>
```

is responsible for generating the HTML links to section anchors, while the line

```
[-#content:BrokenListItemToHTML-]<br>
```

generates the alphabetical list of keyword entries within the section. The template file **generic/BrokenListItemToHTML.tpl** simply refers to another name

```
<br>[-ListItemToHTML-]
```

which is a directory containing different templates for different types of topics. In our case, the relevant file is **ListItemToHTML/Keyword-topic.tpl**, with the following content:

```
<a href="[-<TopicFileReference>[-@topic-id-]-]">[-
   @Name:#content:#escape-html-content
-]</a>
```

Finally, the template **LineToHTML/Group.tpl** receives as input the value of the previously prepared variable *Line*. The template replaces every **Group** element by the HTML link to the corresponding section anchor. The key of the group that matches the leading character of the current section, made ready in the variable *CurrentCharacter*, is presented with a larger font and is not linked. This template is presented below.

```
[-
   #no[-@GroupKey:#match[-$CurrentCharacter-]-]:
   #apply["<a href="#[-@GroupKey:#lowercase-]">[-@GroupKey-]</a>"]
-][-
   #if[-@GroupKey:#match[-$CurrentCharacter-]-]:
   #apply["<font size="+3" color="#000000"><strong>[-@GroupKey-]</strong></font>"]
-]
```

6 Results and Future Work

The OMDE rewrite has been successfully completed in approximately one man-month. This time included the migration of the existing content and verification of the output against the old environment. One measure of the achieved level of separation of concerns, the OmniMark code does not specify a single HTML tag. On the other side of the fence, the templates have proven intuitive enough to be edited by the technical documentation writers, though so far only small modifications have been made.

6-1 Code Size

The HTML-specific module of the 1998 OMDE contained 5827 lines of OmniMark code, as well as 4381 lines of HTML templates. The new publishing system consists of 1884 lines of OmniMark code and 2855 lines of templates. The strict separation of concerns and the clean model of processing logic reuse will hopefully prevent the slow accretion of code that plagued the old system.

The OmniMark code is shorter because it does not contain any presentation-specific logic. Approximately a third of the code is spent on the built-in filter implementations, and these filters are heavily used by the templates: the average number of uses is 13 and the median is 9. Of course, this result can be viewed in two ways: it can be seen as a proof of good factoring of built-in filters, or as an indication of poor factoring of templates. Indeed, the original OMDE templates have been only minimally refactored. The main reason that the line count of templates has decreased is that, in some cases, two or more similar templates have been unified into one in the course of migration. For example, there used to be two separate, but almost identical templates describing the current and legacy libraries. These two templates have been unified into a single template that uses conditional logic to produce two different outputs. It was felt that in this case the benefit of single-sourcing the template in the long term outweighed the disadvantage of the increased template complexity.

Even without the proper factoring of templates, the amount of code reuse is surprisingly high for a project that did not even have code reuse as an explicit goal. This is especially interesting in light of the fact that OMDE has no concept of inheritance. The usual techniques of reuse of the content processing logic, like DITA, focus on

process inheritance [DITA 1998]. The most likely reason for the amount of code reuse is the enforced emphasis on higher-order functions. There has been little research of higher-order functions as a vehicle for code reuse [Wadler 1998], but the observed results certainly match the folklore of the functional programming community.

6-2 Performance

The system builds the entire documentation in HTML form (19 MB in 1892 pages) in approximately three and half minutes on an Athlon 64 X2 3800+ CPU, using a single thread of execution. While the speed could be improved, at this point it is far from being a bottleneck.

In order to publish all index pages, the system partially parses 1461 SGML topic files, allocates 354,902 coroutines, of which at most 39 are used at the same time, and performs 20,943,723 coroutine context switches. To publish all current topics, OMDE performs 15,068 parses of SGML files (some files are parsed more than once), 4,743,939 coroutines allocations (at most 43 simultaneously), and 121,354,197 coroutine switches. The latter task takes three minutes, the bulk of the total processing time. These statistics show that pipelined processing of markup streams has a substantial price, and fast coroutine creation and context switching is of paramount importance [Wilmott 2003]. If we, rather optimistically, assume the thread creation time of 10 microseconds and thread switching time of 1 microsecond [Corsaro 2003], the total overhead for the last task would be over two and half minutes if threads were used instead of coroutines.

6-3 Future Work

At this point the OMDE project, though far from perfect, has achieved its initial goals. The next development iteration will probably concentrate on the generation of alternative output formats: various subset of the documentation for different audiences, as well as output in PDF form. It will be interesting to see what level of template reuse can be achieved when there are multiple output targets.

7 Related Work

There are too many existing template languages, engines and systems to list here. Some of them have various similarities to the one presented here. For example, Smarty [Smarty], Django [Django] and StringTemplate [Parr 2006] all support text filters. Django and StringTemplate allow filter composition. Some of the built-in filters in Django transform structured inputs. Django does not emphasize the use of filters as higher-order functions; it provides a number of control constructs instead. The StringTemplate language is more like OMDE in its minimalism, but its built-in filters do not appear to operate on structured values.

The main difference between OMDE and other template languages appears to be in the choice of the data model. Most template languages use an input data model that matches the data structures available in the host language. Functional languages specifically designed for markup processing, like XSLT [XSLT 1999] and XQuery [XQuery 2007], on the other hand, have a markup-based data model but are not template languages. There has been work on template languages developed on top of XSLT. One example is TemplateXSLT [TemplateXSLT 2002], but at least in this case the template language design is traditional in that it provides explicit control constructs instead of taking advantage of the data model; it is not data-driven.

One significant obstacle in using XSLT as the host language is that its data model is based on trees, not streams. While this is not a problem for small documents (as in the OMDE project presented here) and short pipelines, this model does not scale well because it requires the entire input tree to be constructed in each phase of the pipeline. The benefits of the streaming model are explained in more depth in [Morrison 1994] and [Wilmott 2003].

Functional programming techniques have been applied to markup processing plenty of times. The Haskell library HaXML [HaXML 1999], for example, provides higher order functions (combinators) for processing XML and HTML trees. In the more mainstream context of XSLT, functional programming extensions with higher-order functions have been implemented by FXSLT [Novatchev 2006].

The pipeline processing model has been used for markup processing many times before [Transmorpher 2001] [XPipe 2002] [XML Pipeline 2002]. Most of the current XML pipeline languages rely on components written in a lower-level programming language like XSLT, though some also provide a set of higher-level functions for combining the components. From this perspective, the main novelty presented in this paper is the use of templates for defining the primitive pipeline components instead of XSLT.

The author's previous work [Combinators 2006], while based on the same principles of streaming and higher-order functions, uses a more expressive language because it is not limited to markup processing. The present work restricts the data model in order to simplify the processing language.

8 Conclusion

We have shown how the use of concepts of higher-level functional programming, together with a markup-based data model, can make complex control structures unnecessary in a template language. Furthermore, if we use the same data model both for input and output data, the resulting templates become composable. The concept has been successfully implemented and tested in a production system for publishing technical documentation.

Notes

1. Omission of escaping has many times in the past led to security vulnerabilities. The most notorious variant is probably the vulnerability to SQL injection attacks. **[Su 2006]**

Acknowledgments

I want to thank Jacques Légaré and Helen St. Denis for finding the time to read the paper and point out the problems, and the Stilo Corporation for giving me the time to write it.

Bibliography

[Backus 1978] Backus, J., *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs* Commun. ACM 21, 8 (Aug. 1978), 613-641.

[Baker 1998] Mark Baker, *Designing Microdocument Architecture systems*, http://www.infoloom.com/gcaconfs/WEB/paris98/baker.HTM#, SGML/XML Europe 1998, Paris

[Combinators 2006] Mario Blaæević, *Streaming component combinators*, Extreme Markup Languages, 2006.

[Corsaro 2003] Angelo Corsaro, Douglas C. Schmidt, *The Design and Performance of Real-time Java Middleware*, IEEE Transactions on Parallel and Distributed Systems. Vol. 14, no. 11, pp. 1155-1167. Nov. 2003

[DITA 1998] Priestley, M., *DITA XML: a reuse by reference architecture for technical documentation* In Proceedings of the 19th Annual international Conference on Computer Documentation (Sante Fe, New Mexico, USA, October 21 - 24, 2001). SIGDOC '01. ACM Press, New York, NY, 152-156.

[Django] Lawrence Journal-World, *The Django Template Language*, http://www.djangoproject.com/documentation/templates/, 2007.

[HaXML 1999] Malcolm Wallace, Colin Runciman, *Haskell and XML: Generic Combinators or Type-Based Translation?*, International Conference on Functional Programming, 1999.

[Haskell 2003] S. Peyton Jones, editor, Haskell 98 Language and Libraries: The Revised Report, Cambridge University Press, 2003, http://haskell.org/

[Morrison 1994] John Paul Morrison, *Flow Based Programming*, Van Nostrand Reinhold, 1994.

[Novatchev 2006] Dimitre Novatchev, *Higher-Order Functional Programming with XSLT 2.0 and FXSL*, Extreme Markup Languages, 2006.

[OmniMark 2007] *OmniMark language documentation*, http://developers.stilo.com/docs-extract/html/index.htm

[PHP] *PHP: Hypertext Preprocessor*, http://www.php.net/

[Parr 2004] Parr, T. J., *Enforcing strict model-view separation in template engines*, In Proceedings of the 13th international Conference on World Wide Web (New York, NY, USA, May 17 - 20, 2004). WWW '04. ACM Press, New York, NY, 224-233.

[Parr 2006] Terence Parr, *A Functional Language For Generating Structured Text*, http://www.cs.usfca.edu/~parrt/papers/ST.pdf, University of San Francisco, 2006.

[Smarty] *Smarty: Template Engine*, http://smarty.php.net/

[Su 2006] Su, Z., Wassermann, G., *The essence of command injection attacks in web applications* In Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Charleston, South Carolina, USA, January 11 - 13, 2006). POPL '06. ACM Press, New York, NY, 372-382.

[TemplateXSLT 2002] Jason Diamond, *Template Languages in XSLT*, http://www.xml.com/pub/a/2002/03/27/templatexslt.html XML.com, O'Reilly Media, Inc., 2006

[Transmorpher 2001] Jérôme Euzenat, Laurent Tardif, *XML Transformation Flow Processing*, Extreme Markup Languages, 2001.

[Wadler 1998] Wadler, P., *How to solve the reuse problem? Functional programming*, Proceedings. Fifth International Conference on Software Reuse, 1998.

[Wilmott 2003] Sam Wilmott, *What programming language designers should do to help markup processing*, Extreme Markup Languages, 2003.

[XML Pipeline 2002] *XML Pipeline Definition Language Version 1.0* , http://www.w3.org/TR/2002/NOTE-xml-pipeline-20020228/

[XPipe 2002] *XPipe presentation at XML SIG NY*, http://xpipe.sourceforge.net/BinaryStuff/xpipeny.ppt, 2002.

[XQuery 2007] *XQuery 1.0: An XML Query Language*, http://www.w3.org/TR/xquery/

[XSLT 1999] *XSL Transformations (XSLT) Version 1.0*, http://www.w3.org/TR/xslt/

Composable Templates

Mario Blažević [Senior Software Developer, Stilo International]
