# Streaming component combinators

*We present a small framework for building and reusing components for processing of markup streams. The core of the framework is built around two concepts that, as far as we know, have not been used in this area so far: the concept of splitter components that follow a common interface and contract, and the concept of generic component combinators that can be used to combine arbitrary splitter and filter components into higher-level components.*

Mario Blažević, Senior Software Developer, Stilo International

## Introduction

Streaming, in the context of this paper, means processing data as it is being delivered. There are two reasons why streaming is important for markup processing. One is that streaming applications do not need to load their whole input into main memory at once, and hence achieve better performance and scalability compared to DOM and other technologies dependent on data-structure representations of the input. The other reason is that many applications are naturally expressed in producer-consumer or producer-filter-consumer patterns. The ability to express streaming makes these types of applications much easier to build. See [Morrison 1994] for a more detailed justification of the streaming paradigm.

Today's mainstream programming languages are unfortunately not particularly well-suited to the streaming model [Wilmott 2003]. While it may be easy to implement a simple stream processor, its reuse in the context of a larger streaming application is quite difficult in comparison to the simple reuse of the native programming language concepts like functions or classes.

Another way to support streaming is at the operating system level. Small, reusable streaming applications have traditionally been encouraged in the Unix environment [Raymond 2003]. CMS Pipelines [Hartmann 1992] provide support for more sophisticated streaming applications. At this level, however, the stream processors are treated as black boxes that cannot be type-checked, nor fused and optimized in other ways. The inter-process communication is also much more expensive than language-supported mechanisms would be.

To summarize, neither the programming languages nor the operating systems provide a satisfactory glue logic for combining streaming components together. For these and other reasons, several attempts have been made to provide the missing support for streaming applications in the shape of a *framework*. To date, however, their impact has been limited. One reason for this state of affairs may be that existing frameworks are not modular enough. They provide various reusable components as well as glue logic needed to combine them into a pipeline. However, the generic components that the existing frameworks emphasize are filters. Two components in a filter class can be trivially combined together into a single filter component. There is no simple way to combine more complex components together. Some frameworks allow the user to combine components by connecting their ports together, but this style of component network specification is much more difficult to master than the simple pipelining of two filters.

This paper is a proposal to introduce a new class of generic components that offers more potential for gluing than filters do, but at the same time remains easy to reason about. All components used in this paper, both basic and higher-level component combinators, have been implemented in

OmniMark [OmniMark 2006], version 8. Some of the component implementations have been omitted to conserve space. The same design could be replicated in other general-purpose programming languages. One likely constraint is that the implementation language should support coroutines in some form [Wilmott 2003].

## Related work

The importance of streaming transformations of markup has long been recognized. There has been ongoing work on creating streaming implementations of standardized markup-processing and querying languages, XPath [Peng 2005][Olteanu 2004], XSLT and XQuery [Florescu 2004] [Fegaras 2002] among them. However, the design of these languages is such that no implementation can guarantee streaming behaviour. Other efforts have been focused on defining streaming subsets of the aforementioned languages, as well as specially-designed languages that fully support streaming processing, like OmniMark [OmniMark 2006], Sequential XPath [Desai 2001] and Streaming Transformations for XML [Becker 2003].

Streaming component frameworks seem to have been reinvented several times. Most of these frameworks, however, are designed to work on coarse-grained filter components. Some examples are XPipe [XPipe 2002], XML Pipeline [XML Pipeline 2002], Cocoon [Apache 2006], and iFlow [Lui 2000]. Possible exceptions are STnG [Krupnikov 2003], as well as THREADS [Morrison 1994] which can apply components to arbitrary parts of the framework input. However, none of the listed frameworks seem to try to formally classify the components into abstract classes, nor define any generic component combinators that would operate on instances of component classes.

On the other hand, plenty of work has been done on combinator frameworks in general, especially in the functional programming language community. The *parser* component combinators [Hutton 1996] are closely related to this work. A nice implementation of a parser combinator library is Parsec [Leijen 2001]. The main difference between a parser combinator framework and a streaming component combinator framework, such as the one presented here, is that parser frameworks are streaming only on their input side. On the output side, they produce a well-defined and structured result from the input source, typically a parse tree. This is possible only because a parser deals with highly structured and constrained input. Once some input is parsed and the result constructed, the input is no longer necessary. In contrast, a streaming component combinator framework has to deal with input that is only weakly structured; in many cases the only meaningful result of processing any single component is just a rearrangement of the component's input.

There are also combinator frameworks for XML document processing, HaXML [Wallace 1999] for example. While many of the combinators defined by HaXML resemble ours, there are two important differences: HaXML combinators operate exclusively on filters and *ad hoc* Haskell function types, and HaXML operates on a parsed document tree and therefore is not streaming. The latter constraint appears to be shared by all XML combinator frameworks implemented in Haskell today. The XPath implementation in Scheme, SXPath [Lisovsky 2003], provides combinators that operate on nodeset-transforming functions, which are also not designed for streaming.

The most recent work [Shivers 2006] finally applies the combinator approach to streaming components by using continuation passing to transfer control. The early results are promising, but most attention is still devoted to simple linear pipelines of filter components.

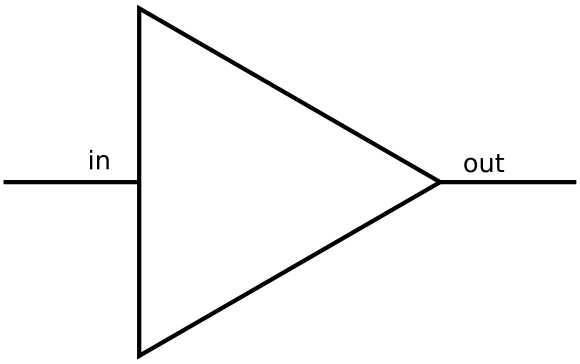
# Streaming component types

## Filters

A filter is a streaming component with one data source and one data sink. The concept of streaming filter components has been recognized for some time, and is a part of programmers' vocabulary in many areas. A filter component takes a data stream as its input, transforms it, and produces another data stream as its output. The best-known example of filters is probably the Unix piping model [Raymond 2003].

A filter component has a responsibility to consume its whole input. It is not allowed to simply terminate in the middle of processing of its input. If one filter in a pipeline terminates, the whole pipeline dies with it.

In OmniMark, a filter component can be represented as an abstract data type with a single operation *apply-filter*, illustrated below in figure 1:




```
export record filter

export dynamic overloaded function
   apply-filter value filter        filter
            from value string source filter-source
            into value string sink   filter-sink
as
   not-reached message "The base function apply-filter must be overridden!"
```

**Figure 1. "Abstract filter component"**

A concrete filter implementation must extend the abstract *filter* type and override its *apply-filter* method. Several concrete filter examples are given next.

## Basic filters

- **as-is** is a simple filter that passes its input through unmodified.

```
; record declaration
declare record identity-filter extends filter
```

```
; constructor definition
export filter function
   as-is
as
   return new identity-filter {}

; apply-filter method overriding
define overriding function
   apply-filter value identity-filter filter
           from value string source   filter-source
           into value string sink     filter-sink
as
   put filter-sink filter-source
```

- The **replace-by** filter is parametrized by a string parameter at creation time. This filter replaces its whole input by the parameter.

```
; record declaration
declare record constant-filter extends filter
   field string replacement

; constructor definition
export filter function
   replace-by value string replacement
as
   local constant-filter result

   set result:replacement to replacement
   return result

; apply-filter method overriding
define overriding function
   apply-filter value constant-filter filter
           from value string source   filter-source
           into value string sink     filter-sink
as
   put filter-sink filter:replacement
   put #suppress filter-source
```

- **suppress** filter suppresses all input it receives. It is equivalent to *replace-by ""*.
- **error** filter reports an error if any input reaches it.
- **prepend** filter passes its input unmodified, except for prepending a given constant string parameter before it.
- **append** filter passes its input unmodified, except for appending a given constant string parameter to its end.
- **include-file** treats its whole input as a URL or a file path and outputs the content of the file.

## Markup filters

The filters listed in the previous section are generic, in the sense that they can be applied to any form of character input stream. But if we restrict our attention to inputs in a particular format, we can provide many more interesting filters. The more specific the input is, the more sophisticated the possible processing components can get. For example, we can apply the following filters to any input in XML format:

- **xml-escape** escapes the XML markup so it can be embedded as an XML element content. It replaces ">" characters by ">" etc. Its inverse is **xml-unescape** which performs the opposite operation, by expanding all escaped XML character entities.
- **xml-rename-element** renames all top-level occurrences of any XML element in its input to the given name.
- **xml-rename-attribute** renames all top-level elements' attributes with the given name to the other given attribute name.
- **xml-add-attribute** adds an attribute with a given name and value to all top-level elements in its input.
- **xml-remove-attribute** removes all top-level element attributes in its input that match the given attribute name.

The inputs and outputs of XML-specific components are not well-formed XML documents, but rather streams of well-formed mixed content.

## Buffering filters

There is a large class of filters that have to buffer their whole input before they can produce any result. These are not *streaming* components in any sense, but since they expose their functionality through a streaming interface they can still be used in a streaming component framework. The trick is to restrict their usage to the parts of the stream which must be processed in this way. Unfortunately, the burden of deciding which parts those are is on the user.

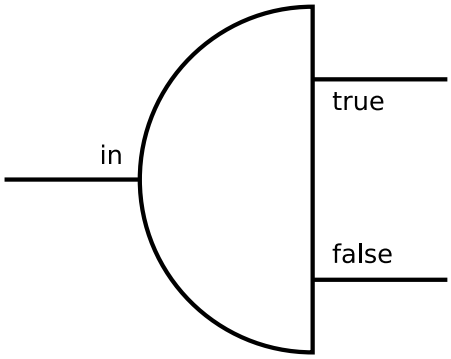
- **xml-sort-by-attribute** sorts the top-level elements in its input by the value of the attribute with the given name.
- **xml-sort-by-content** sorts the top-level elements in its input by the value of their content.
- **character-count** counts all characters in its input and outputs the number in decimal form.

# Splitters

A splitter is a streaming component with one data source and two sinks. A pure splitter component is not supposed to modify its input in any way; instead, it streams portions of its input stream to either of its two outputs. Every bit from the input stream has to be present in one or the other of the output streams. Secondly, the outputs have to be properly interleaved. Another way of stating the invariants is to say that if the two sinks of any pure splitter are connected together, the component acts as an identity transform.

The concept of splitters, together with the operations that can be applied to them, is the only new technical contribution this paper has to offer. The streaming component frameworks have been researched and built before, but the emphasis has mostly been on filters. That is not to say that no framework contained splitter components before. The previous work on streaming pipeline frameworks has mostly seen splitters as large, *ad hoc* components that must be provided by the user.

In OmniMark, a splitter component can be represented as an abstract data type with a single operation *split* that takes one data source and two sinks as arguments:




```
export record splitter

export dynamic string sink function
   split (value splitter    sp,
          value string sink true,
          value string sink false)
as
   not-reached message "The base method split must be overridden!"
```

**Figure 2. Abstract splitter component**

The concrete splitter instances must extend the base *splitter* type and override the method *split*. In the examples that follow, the OmniMark code will be elided for brevity.

## Basic splitters

- **all-true** and **all-false** splitters simply pass their entire input to the *true* and *false* sink, respectively.
- The **whitespace** splitter passes all whitespace to its *true* sink, and the rest of the input to its *false* sink.
- The **lines** splitter passes the contents of every line in its input to the *true* sink, and all line-end characters to the *false* sink.

- The **substring** splitter is parametrized by a constant string value. Whenever it encounters the given string in its input, it passes it to the *true* sink. All the rest of the input goes to the *false* sink.

```
; record declaration
declare record substring-splitter extends splitter
   field string substring

; constructor definition
export splitter function
   substring value string substring
as
   local substring-splitter result

   set result:substring to substring
   return result

; split method overriding
define overriding string sink function
   split (value substring-splitter self,
          value string sink        true,
          value string sink        false)
as
   repeat scan #current-input
   match ~self:substring
      put true self:substring
   match any => one
      put false one
   again
```

- **prefix** and **suffix** are similar to the *substring* splitter, except they look for the given string only at the beginning and at the end of their input, respectively.

## Markup splitters

The observation on the generic and purpose-specific filters also holds true for splitters: as we restrict the inputs to be more specific, the splitters we can apply to them become more sophisticated.

- **xml-element** scans the input for well-formed XML elements. Elements are passed to the *true* sink together with their content, and all the intervening data goes to the *false* sink. If there is no intervening data between two consecutive elements, this splitter sends an empty XML comment to the *false* sink as intervening data. Therefore, this is not a pure splitter: it generates some data that was not present in its input. This behaviour makes the splitter usable for chunking consecutive elements, as will be seen later. When used with the other element-processing components, as is usually the case, the extra empty comments are irrelevant.
- **xml-element-content** behaves similarly to the *xml-element* splitter, except it strips the start and end tags off the top-level elements, and sends only their content to the *true* sink. The top-level element tags, together with any intervening data content, go to the *false* sink.

- **xml-element-named** is parametrized by an element name, and optionally with a namespace. It behaves like the *xml-element* splitter, but it sends to its *true* sink only those top-level elements whose name matches the given argument name.
- **xml-attribute-named** is parametrized by an attribute name, and optionally with a namespace. If it finds an element attribute matching the given name among the top-level elements in its input, it sends it to the *true* sink. Everything else goes to the *false* sink.
- **xml-attribute-value** expects a list of attributes as its input. It sends attribute values, with the quote delimiters stripped, to its *true* sink, and the rest of the input to its *false* sink.

# Streaming component combinators

A library of streaming components like those listed in the previous section can help perform various basic tasks, but we call them *components* because the goal is to use them together as building blocks for assembling more sophisticated stream-processing tools.

One way to combine streaming components is to wire their ports together; for examples see [Morrison 1994] and [XML Pipeline 2002]. This method directly corresponds to the way pipelines are usually represented graphically, by drawing lines between components' ports. While a specification written in this style can express any possible configuration of components, the technique does not scale well: the number of possible connections grows as the square of the number of all ports of all components present in the network.

The other common approach is to use combinators. Component combinators operate on whole components instead of their ports, much in the same way that combinatory logic [Curry 1958] operates on whole functions instead of their arguments and results. An example of this specification method in streaming frameworks is [Krupnikov 2003], but the best known streaming combinator is the Unix pipe operator.

While the wiring approach is applicable to any number of components of any kind, a combinator can be applied only to specific kinds of components. This constraint leads to two design imperatives:

- All basic components should be classified by what combinators can be applied to them, and the number of different component classes should be kept to a minimum. The more different component classes there are, the more different combinators are needed to operate on them.
- The result of a combinator application is another component. This combined component should always fall into one of the component classes of the framework. That way we can keep applying the same combinators to the intermediate results, producing more and more complex components.

The framework presented here classifies all components into two classes: filters and splitters. Every basic component is either a splitter or a filter, the only arguments of every combinator are splitters and filters, and the result of each combinator application is either a filter or a splitter component.

## Filter combinators

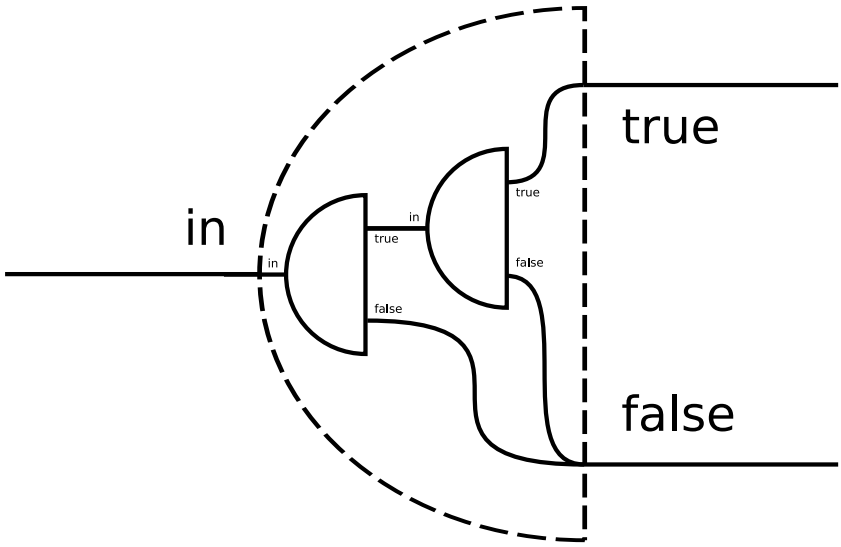
- >> The most obvious and well-known combinator of streaming components is the pipe operator. It takes two filters and combines them together into a single filter component. The resulting component acts as a composition of the two argument filters: it passes the input to one filter, the output of that filter is passed to the other filter, and its output becomes the output of the combined component.
- **join** combinator also combines two filters into a single filter. However, this combinator arranges the two filters in parallel. The input of the resulting filter is replicated to both component filters in parallel, and the output of the resulting filter is a concatenation of the two component filters' outputs.

  To do its job, the join combinator uses two non-standard components. The **tee** component, known from the Unix environment, replicates its input into two parallel streams. This component is neither a filter nor a splitter, so no generic component combinator can use it. Another non-standard Unix-style component is **cat** taking two (or more) inputs and concatenating them in order. Again, this component cannot be used by generic component combinators. However, in place of these two components we can provide the *join* combinator that combines two filters into another filter component. The combined component internally uses *tee* to replicate its input to both of the filters it contains and then *cat* to concatenate the filters' outputs. The result behaves as an ordinary filter and can be used by other combinators. Notice, however, that the result filter of join is not completely streaming, as it has to buffer the output of the second component filter until the end of the input.

## Pseudo-logical splitter combinators

A pure splitter component can be thought of as a representation of a logic predicate. The portion of the input that the splitter sends to one of its two sinks is taken to satisfy the predicate, and the input that goes to its other sink does not. Having this picture in mind, the following splitter combinators become obvious.

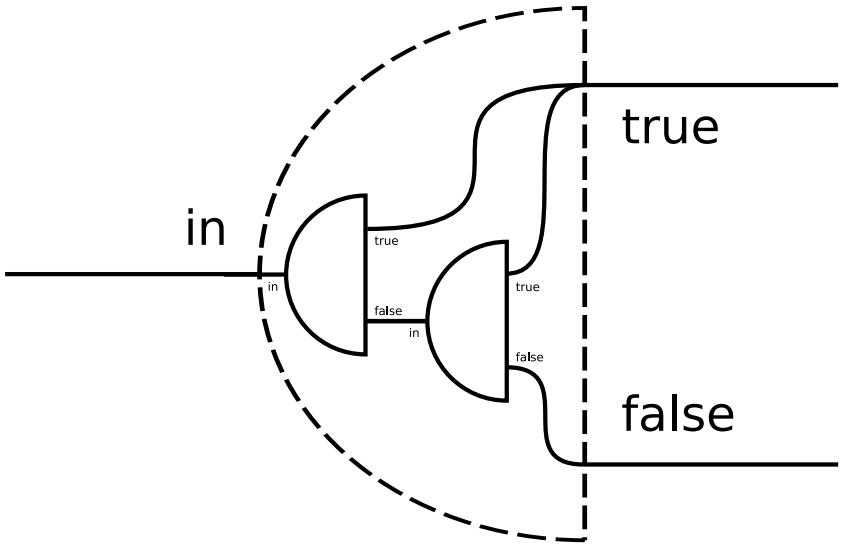
- The **!** (not) operator simply reverses the outputs of the argument splitter. In other words, data that the argument splitter sends to its *true* sink goes to the *false* sink, and vice versa.
- The **>&** operator sends the *true* sink output of its left operand to the input of its right operand for further splitting. Both operands' *false* sinks are connected to the *false* sink of the combined splitter, but any piece of input to get into the *true* sink of the combined component data must pass both splitters. Figure 3 illustrates this.




```
; record declaration
declare record and-splitter extends splitter
   field splitter left
   field splitter right

; constructor definition
export overloaded splitter infix-function
      value splitter left
   >&
      value splitter right
as
   local and-splitter result

   set result:left to left
   set result:right to right
   return result

; split method overriding
define overriding string sink function
   split (value and-splitter s,
          value string sink  true,
          value string sink  false)
as
   using output as split (s:left, split (s:right, true, false), false)
      output #current-input
```

**Figure 3. >& splitter combinator**

- The >| splitter combinator is a mirror image of >& combinator, as illustrated by figure 4. Its input can get to its *false* sink only by going through both argument splitters' *false* sinks.




```
; ... skipping the record and constructor declaration
; split method overriding
define overriding string sink function
   split (value or-splitter s,
          value string sink true,
          value string sink false)
as
   using output as split (s:left, true, split (s:right, true, false))
      output #current-input
```

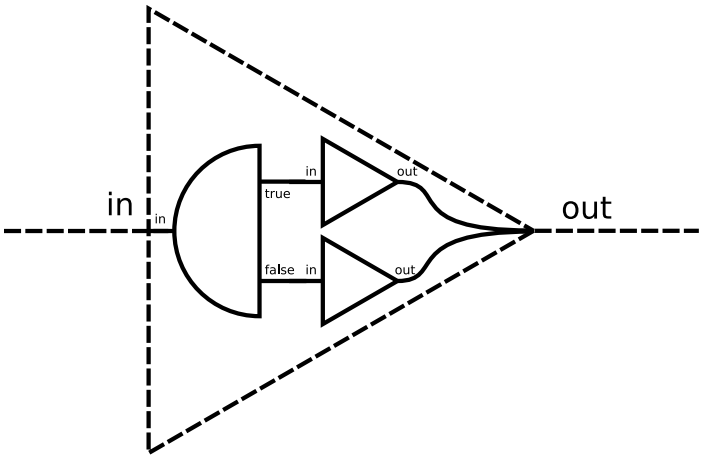
**Figure 4. >| splitter combinator**

Note that the >& and >| operators are not exact equivalents of the corresponding Boolean *and*/*or* operators. In particular, they are commutative only when their two argument splitters agree on what forms a unit of input, *i.e.*, when the input stream is a uniform sequence of records. Most markup splitters do not satisfy this property. For example, the expression *xml-element-named "A" >& xml-element-content >& xml-element-named "B"* is an equivalent of the XPath expression A/B, which is obviously not commutative.

## Flow-control combinators

Since approximating a splitter as a logic predicate ( *i.e.*, a boolean expression) turned out to be productive, we can try digging deeper into our intuition for useful metaphors. For example, we can see a filter component as an imperative statement. A filter modifies the stream that flows through it, much like an imperative statement that modifies the state surrounding it. The piping operator >> would then be an equivalent of a statement sequencing operator. Having both imperative statements and Boolean expressions, the natural thing to look for next would be an equivalent of the flow-control statements.

- The **if** combinator takes a splitter and two filters and combines them into a single filter component, as shown in figure 5. The resulting filter applies one argument filter to one portion of the input and the other filter to the other portion of input, depending on where the splitter routes the data.




```
; apply-filter method overriding
define overriding function
   apply-filter value if-filter     filter
           from value string source filter-source
           into value string sink   filter-sink
as
   using output as split (filter:splitter,
                          filter:left >> filter-sink,
                          filter:right >> filter-sink)
      output filter-source
```

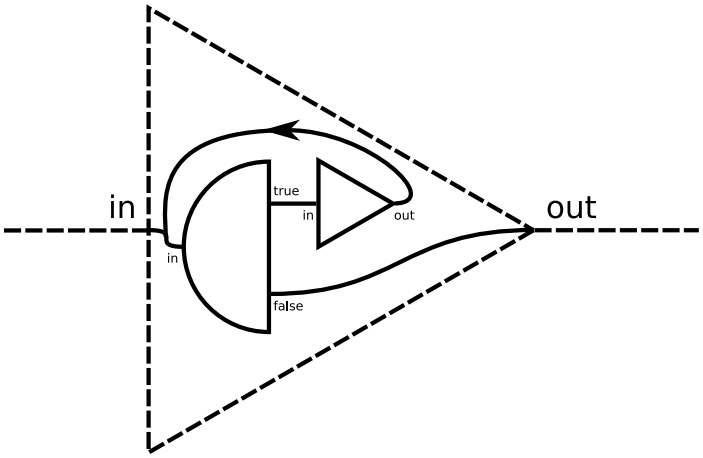
**Figure 5. if combinator**

The combinators **where** and **unless** are simpler versions of the *if* combinator, having one of the two filters be *as-is*. The **select** combinator uses the *as-is* and *suppress* filters as the *true* and *false* sinks, respectively. In effect, the *select* combinator converts a splitter into a filter which retains only the *true* portion of the input. The three combinators can be more formally defined as follows:

- *splitter* **where** *filter* ⟶ if *splitter* then *filter* else as-is
- *splitter* **unless** *filter* ⟶ if *splitter* then as-is else *filter*
- **select** *splitter* ⟶ suppress unless *splitter*

A combinator language is point-free by definition, so we cannot use component names or labels to achieve looping and recursion. If we want our component networks to be able to form loops, we need more flow-control combinators.

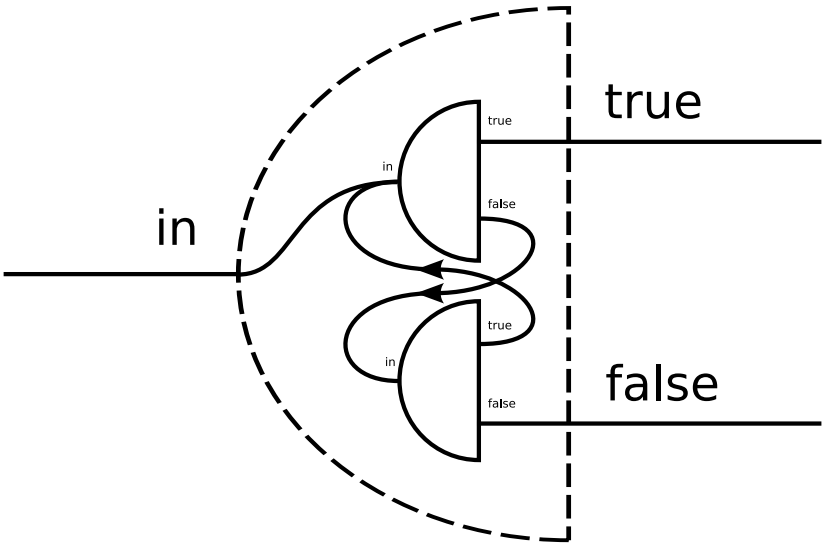
- The recursive combinator **while**, illustrated in figure 6, feeds the *true* sink of the argument splitter back to itself, filtered by the argument filter. The *false* sink of the splitter is passed on unmodified. The combinator **until** is the mirror image of *while*.




```
; apply-filter method overriding
define overriding function
   apply-filter value while-filter  self
           from value string source filter-source
           into value string sink   filter-sink
as
   do scan filter-source
   match lookahead any
      using output as split (self:splitter,
                             self:filter >> self >> filter-sink,
                             filter-sink)
         output #current-input
   done
```

**Figure 6. while combinator**

- The recursive combinator **nested** combines two splitters into a mutually recursive loop acting as a single splitter. The *true* sink of one of the argument splitters and *false* sink of the other become the *true* and *false* sinks of the loop. The other two sinks are bound to the other splitter's source. This particular combinator does not have a corresponding flow-control equivalent in the world of programming languages. The reason for this is that the use of *nested* only makes sense on hierarchically structured streams. If we gave it some input containing a flat sequence of records, and assuming both component splitters are deterministic and stateless, a record would either not loop at all or it would loop forever.




```
; split method overriding
define overriding string sink function
   split (value nested-splitter self,
          value string sink     true,
          value string sink     false)
as
   do when #current-input matches (lookahead any)
      using output as split (self:upper-splitter,
                             true,
                             split (self:lower-splitter,
                                    split (self, true, false),
                                    false))
      repeat scan #current-input
      match any => one
         output one
      again
   done
```

**Figure 7. nested combinator**

## Active combinators

Each of the combinators defined so far builds a streaming network out of its argument components, and then lets the network do its job. The combinator itself plays no role after the initial configuration. This property guarantees that the combinator is completely generic: the type of input that can be processed by the network is constrained only by its components, not by the combinators applied to them.

The combinators defined next are different; they are still generic enough to be used on any kind of input, but they also actively process the input stream. They achieve these seemingly contradictory goals by using their splitter argument to determine the structure of the input. Every

contiguous portion of the input that gets passed to one or the other sink of the splitter is treated as one section in the logical structure of the input stream. What is done with the section depends on the combinator, but the sections, and therefore the logical structure of the stream, are determined by the argument splitter alone.

- The combinator **first** takes a splitter as an argument and returns another splitter. The result behaves the same as the argument splitter up to and including the first portion of the input which goes into the argument's *true* sink. All input following the first *true* portion goes into the *false* sink.
- The result of the combinator **last** is a splitter which directs all input to its *false* sink, up to the last portion of the input which goes to its argument's *true* sink. That portion of the input is the only one that goes to the resulting component's *true* sink.

  The splitter returned by the combinator *last* has to buffer the previous two portions of its input, because it cannot know if a *true* portion of the input is the last one until it sees the end of the input or another portion succeeding the previous one.

- The **foreach** combinator is similar to the combinator *if* in that it combines a splitter and two filters into another filter. However, in this case the filters are re-instantiated for each consecutive portion of the input as the splitter chunks it up. Each contiguous portion of the input that the splitter sends to one of its two sinks gets filtered through the appropriate argument filter as that filter's whole input. As soon as the contiguous portion is finished, the filter gets terminated.
- The **having** combinator combines two pure splitters into a pure splitter. Again, one splitter is used to chunk the input into contiguous portions. Its *false* sink is routed directly to the *false* sink of the combined splitter. The second splitter is instantiated and run on each portion of the input that goes to first splitter's *true* sink. If the second splitter sends any output at all to its *true* sink, the whole input portion is passed on to the *true* sink of the combined splitter, otherwise it goes to its *false* sink.
- The **having-only** combinator is analogous to the *having* combinator, but it succeeds and passes each chunk of the input to its *true* sink only if the second splitter sends no part of it to its *false* sink.

Note that the combinators *having* and *having-only* must buffer each complete chunk of input until the second splitter tests it.

One reason buffering matters is that flow-control combinators require their argument components to agree on which parts of the stream get buffered and at which point they get flushed. If one of the subcomponents buffers a part of its input and the other does not, the output of the combined component may be improperly interleaved. The following filter expression is one such example:

```
if xml-element-content then as-is else select last all-true
```

Both branches of the expression produce identical output, so the whole expression may seem equivalent to *as-is*, but because all the element tags get buffered by *last all-true* they will be output after the element content. We can synchronize the two branches by using the *foreach* combinator instead of *if*.

# Examples

Let us try to define some higher-level components as a test of the framework's expressivity.

- The **matching** combinator takes a splitter *selector* and a string constant *value* and returns a splitter that compares each portion of the input passed by the selector against the value:

  ```
  selector matching value ⟶ selector having-only prefix value
  ```

  or, in OmniMark syntax:

  ```
  export splitter infix-function
        value splitter selector
     matching
        value string value
  as
     return selector having-only prefix value
  ```

- The **attribute-is** splitter, parametrized by two strings, *name* and *value*, passes all top-level elements that have an attribute with the same name and value.

  ```
  name attribute-is value
     ⟶ xml-element having ((xml-attribute-named name >& xml-attribute-value)
                              matching value)
  ```

- The **xml-element-count** filter takes XML input and outputs the number of all top-level elements.

  ```
  xml-element-count
     ⟶ (foreach xml-element then replace-by "." else suppress) >> character-count
  ```

- The **descendant-element-named** splitter component finds all nested elements with the given name:

  ```
  descendant-element-named name
     ⟶ nested (xml-element-named name) in xml-element-content
  ```

- The **text-content** splitter extracts the text content of all top-level elements in its input:

  ```
  text-content ⟶ xml-element-content >& !xml-element
  ```

- The next example illustrates the use of the *join* combinator. For any *file-path* element in the input stream, whose content is a slash-delimited file path, the component adds a corresponding base file name enclosed in *file-name* element tags.

  ```
  foreach descendant-element-named "file-path"
  then as-is join ((select xml-element-content)
                   >> (select last (!substring "/"))
                   >> (prepend "<file-name>")
                   >> (append "</file-name>"))
  ```

- As the final example, consider the following XQuery expression:

```
<references>
  {
    for $r in //book[.//keyword[text() = "filter"]]
    return <reference>{$r/title/text()}</reference>
  }
</references>
```

  Assuming there are no book elements nested within each other, this query can be rewritten using the streaming combinators in the following fashion:

```
(
  foreach (descendant-element-named "book"
           having (descendant-element-named "keyword"
                   >& text-content matching "filter"))
  then (select (descendant-element-named "title") >& xml-element-content)
       >> (prepend "<reference>") >> (append "</reference>")
  else suppress
)
  >> (prepend "<references>") >> (append "</references>")
```

# Conclusion and future work

We have demonstrated that the concept of splitter and filter components, coupled with the power of generic component combinators, is expressive enough to build new filters and new splitters from a small base of predefined basic components and component combinators.

In the future, we hope to use the presented framework in real-world applications. More practice is needed to grow the library of basic components and combinators.

We can see that all components communicate with each other through simple data streams, with no out-of-bounds control signals whatsoever. While this feature of the framework simplifies manipulation and reasoning about the component networks, it is a constraint in some cases. In particular, one must take care when using buffering components: If they disagree on the structure of the input stream the output of the network can be improperly interleaved. A solution that could discover a problem of this kind by a static analysis of the component network would be greatly preferable to synchronization signals.

Another area of research would be concerned with various transformations of the component networks. One possibility is automatic conversion of XPath and XQuery expressions into a streaming component representation. At the other end, a built-up component pipeline, or at least parts of it, could be compiled into more efficient code in OmniMark or another host language. And in between, the component network could be transformed either at the high level, using a graph rewriting system, or at the lower level, by fusing multiple components into a single one.

Though the current implementation is in OmniMark, the only features the framework requires from the implementation language are abstract data types and some form of coroutine support. One interesting possibility would be to implement the same framework in Haskell, since the existing implementations of parser combinator libraries in Haskell have proven quite successful.

# Acknowledgements

I want to thank Jacques Légaré for his comments, Helen St. Denis, Joe Gollner and Norbert Winklareth for finding the time to read the paper, and the rest of the team at Stilo for providing the initial motivation for the work.


# Bibliography


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

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

**[Raymond 2003]**Eric Steven Raymond, *The Art of Unix Programming*, http://www.faqs.org/docs/artu/index.html, 2003.

**[Hartmann 1992]**John P. Hartmann, IBM Denmark, *CMS Pipelines Explained* http://vm.marist.edu/~pipeline/pipjarg.pdf, 1992, revised 1997.

**[OmniMark 2006]***OmniMark language documentation*, http://developers.omnimark.com/documentation/index.htm

**[Peng 2005]** Feng Peng, Sudarshan S. Chawathe, *XSQ: A streaming XPath engine*, ACM Transactions on Database Systems (TODS), **30**:2, 2005.

**[Olteanu 2004]**Dan Olteanu, Tim Furche, François Bry, *An efficient single-pass query evaluator for XML data streams*, Proceedings of the 2004 ACM symposium on Applied Computing**627 - 631**, 2004.

**[Florescu 2004]**Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, J. Carey, Arvind Sundararajan, *The BEA streaming XQuery processor*, The VLDB Journal *13*:3, 2004.

**[Fegaras 2002]**Leonidas Fegaras, David Levine, Sujoe Bose, Vamsi Chaluvadi, *XML query processing: Query processing of streamed XML data*, Proceedings of the eleventh international conference on Information and knowledge management, 2002.

**[Desai 2001]** Arpan Desai, *Introduction to Sequential XPath* , XML 2001.

**[Becker 2003]** Oliver Becker, *Extended SAX Filter Processing with STX*, http://stx.sourceforge.net/, Extreme Markup Languages, 2003

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

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

**[Apache 2006]**http://cocoon.apache.org/2.1/overview.html

**[Lui 2000]** Andrew Kwok-Fai Lui, Mark W. Grigg, Michael J. Owen, T. Andrew Au, *iFlow (poster session): a data streaming application framework based on a uniform abstraction*, Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum), 2000.

**[Krupnikov 2003]** K. Ari Krupnikov, *STnG - a Streaming Transformations and Glue framework*, Extreme Markup 2003.

**[Hutton 1996]** Graham Hutton, Erik Meijer, *Monadic Parser Combinators*, Technical report NOTTCS-TR-96-4. Department of Computer Science, University of Nottingham, 1996.

**[Leijen 2001]** Daan Leijen, *Parsec, a fast combinator parser*, http://www.cs.uu.nl/people/daan/parsec.html, University of Utrecht, Dept. of Computer Science, 2001

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

**[Lisovsky 2003]** Kirill Lisovsky, Dmitry Lizorkin, *XML Path Language (XPath) and its functional implementation SXPath*, Russian Digital Libraries Journal, Year 2003, Issue 4

**[Shivers 2006]** Olin Shivers, Matthew Might, *Continuations and Transducer Composition*, PLDI 2006.

**[Curry 1958]** HB Curry, R Feys, W Craig, *Combinatory logic*, North-Holland, 1958.

# Biography

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

The author has a master's degree in computer science from University of Novi Sad, Yugoslavia. Since moving to Canada in 2001, he's been working for OmniMark Technologies, later acquired by Stilo International plc., mostly in the area of markup processing and on development of the OmniMark programming language.