Latte Library

Bob Glickstein

1 Introduction

This document describes the programming interface to the Latte library. Latte is the Language for Transforming Text. With the Latte library it is possible to write programs that process Latte-formatted documents, e.g. to translate them into other formats. One such translator program, `latte-html`, converts Latte files into HTML files. It comes with Latte, and its implementation will serve as an instructional tool throughout this manual.

Regrettably, this document is not yet complete. Until it is, programmers interested in using the Latte library should contact `latte-dev@zanshin.com` for information and assistance.

2 Latte processing overview

To process a file of Latte input, the caller creates a *reader* and a *visitor*. The reader is responsible for parsing the input into a variety of data structures and evaluating those structures. The visitor is responsible for performing the desired processing on the results.

Each top-level expression recognized by the reader is immediately evaluated and then "visited" by the visitor. (The action of the visitor in `latte-html` is to evaluate the expression, adjust the resulting text according to certain HTML rules, and then output it.)

3 Latte_Reader

Latte_Reader (`istream` &*stream*, `const shstring` &*filename*) Constructor
Latte_Reader (`istream` &*stream*, `const shstring` &*filename*, `Latte_Activation` &*activation*) Constructor

Constructs a new `Latte_Reader`. Input is from *stream*. The name of the stream is *filename* (used in error messages and to set the Latte variable `\__FILE__`). The optional argument *activation* contains variable bindings; if omitted, a new global activation is created and populated with some intrinsic global definitions (presently only `\__latte-version__`).

`void` **install_standard_definitions** () `Latte_Reader` member fn

Populates the reader's activation with the standard intrinsic variable definitions, primarily the definitions of Latte's built-in functions.

`void` **process** (`Latte_Visitor` &*visitor*) `Latte_Reader` member fn

Consumes the input and processes it using *visitor*.

4 Latte_Obj

Calling `Latte_Reader::process()` creates a series of `Latte_Obj` trees that are then handed to a `Latte_Visitor`. `Latte_Obj` is an abstract base class for a rich variety of subclasses.

`Latte_Obj` has a collection of virtual functions for testing which subclass an object belongs to.

`Latte_Assignment *` **as_assignment** () `Latte_Obj` virtual member fn
`const Latte_Assignment *` **as_assignment** () `const`
If `this` is a `Latte_Assignment`, returns it, else returns `NULL`.

`Latte_Boolean *` **as_boolean** () `Latte_Obj` virtual member fn
`const Latte_Boolean *` **as_boolean** () `const`
If `this` is a `Latte_Boolean`, returns it, else returns `NULL`.

`Latte_Group *` **as_group** () `Latte_Obj` virtual member fn
`const Latte_Group *` **as_group** () `const`
If `this` is a `Latte_Group`, returns it, else returns `NULL`.

`Latte_List *` **as_list** () `Latte_Obj` virtual member fn
`const Latte_List *` **as_list** () `const`
If `this` is a `Latte_List`, returns it, else returns `NULL`.

`Latte_Nested *` **as_nested** () `Latte_Obj` virtual member fn
`const Latte_Nested *` **as_nested** () `const`
If `this` is a `Latte_Nested`, returns it, else returns `NULL`.

`Latte_Operator *` **as_operator** () `Latte_Obj` virtual member fn
`const Latte_Operator *` **as_operator** () `const`
If `this` is a `Latte_Operator`, returns it, else returns `NULL`.

`Latte_Param *` **as_param** () `Latte_Obj` virtual member fn
`const Latte_Param *` **as_param** () `const`
If `this` is a `Latte_Param`, returns it, else returns `NULL`.

`Latte_Str *` **as_str** () `Latte_Obj` virtual member fn
`const Latte_Str *` **as_str** () `const`
If `this` is a `Latte_Str`, returns it, else returns `NULL`.

`Latte_Tangible *` **as_tangible** () `Latte_Obj` virtual member fn
`const Latte_Tangible *` **as_tangible** () `const`
If `this` is a `Latte_Tangible`, returns it, else returns `NULL`.

`Latte_VarRef *` **as_varref** () `Latte_Obj` virtual member fn
`const Latte_VarRef *` **as_varref** () `const`
If `this` is a `Latte_VarRef`, returns it, else returns `NULL`.

`Latte_WsNode *` **as_wsnode** () `Latte_Obj` virtual member fn
`const Latte_WsNode *` **as_wsnode** () `const`
If `this` is a `Latte_WsNode`, returns it, else returns `NULL`.

Every Latte object has a boolean value and a numeric value.

`bool` **bool_val** () `const` `Latte_Obj` virtual member fn

Returns the boolean value of `this`. By the default implementation, the boolean value of every Latte object is true except for an empty `Latte_List` or `Latte_Group`, and except for a false-valued `Latte_Boolean`. (This does not prevent defining a new subclass that overrides `bool_val()` and creates new false Latte values.)

`Latte_Number_t` **numeric_val** () `const` `Latte_Obj` virtual member fn

Returns the numeric value of `this`. By the default implementation, the numeric value of every Latte object is zero except for `Latte_Str` objects whose strings can be parsed as numbers. (This does not prevent defining a new subclass that overrides `numeric_val()` and creates new Latte objects with non-zero numeric values.)

The type `Latte_Number_t` is either `long` or `double`, depending on an option given when Latte is compiled. If the preprocessor symbol `ENABLE_FLOATING_POINT` (in '`latte-conf.h`') is defined, then `Latte_Number_t` is `double`, otherwise it's `long`.

`Refcounter<Latte_Obj>` **eval** (`Latte_Activation` &*activation*) `Latte_Obj` member fn

Evaluates `this` in the scope of *activation* and returns the result. Every Latte object evaluates to some Latte object; by the default implementation, most simply evaluate to themselves.

Note, `eval()` is not a virtual member function. To define a new subclass with different evaluation rules, override `do_eval()`, not `eval()`.

`bool` **self_evaluating** () `const` `Latte_Obj` virtual member fn

Returns true if `this` will definitely evaluate to itself when `eval()` is called; returns false if `this` *may not* evaluate to itself.

`void` **visit** (`Latte_Visitor` &*visitor*) `Latte_Obj` virtual member fn

"Visit" `this` with *visitor*. This calls `visitor.visit_`*foo*`()` where *foo* is one of `assignment`, `boolean`, `closure`, `group`, `list`, `nested`, `operator`, `param`, `varref`, `wsnode`, and `str`, depending on which `Latte_Obj` subclass `this` belongs to.

4.1 Latte_Tangible

4.2 Latte_Nested

4.3 Other Latte_Obj subclasses

5 Latte_Visitor

`void` **visit_assignment** (`Latte_Assignment` &*assignment*) `Latte_Visitor` virtual member fn

Called by `Latte_Obj::visit()` when the `Latte_Obj` is actually a `Latte_Assignment`. The default implementation does nothing.

`void` **visit_boolean** (`Latte_Boolean` &*boolean*) `Latte_Visitor` virtual member fn

Called by `Latte_Obj::visit()` when the `Latte_Obj` is actually a `Latte_Boolean`. The default implementation does nothing.

`void` **visit_closure** (`Latte_Closure` &*closure*) `Latte_Visitor` virtual member fn

Called by `Latte_Obj::visit()` when the `Latte_Obj` is actually a `Latte_Closure`. The default implementation does nothing.

`void` **visit_group** (`Latte_Group` &*group*) `Latte_Visitor` virtual member fn

Called by `Latte_Obj::visit()` when the `Latte_Obj` is actually a `Latte_Group`. The default implementation recursively calls `Latte_Obj::visit()` on each member of *group* in order.

`void` **visit_list** (`Latte_List` &*list*) `Latte_Visitor` virtual member fn

Called by `Latte_Obj::visit()` when the `Latte_Obj` is actually a `Latte_List`. The default implementation recursively calls `Latte_Obj::visit()` on each member of *list* in order.

`void` **visit_nested** (`Latte_Nested` &*nested*) `Latte_Visitor` virtual member fn

Called by `Latte_Obj::visit()` when the `Latte_Obj` is actually a `Latte_Nested`. The default implementation calls `Latte_Obj::visit()` on the `Latte_Obj` contained within *nested*.

`void` **visit_operator** (`Latte_Operator` &*operator*) `Latte_Visitor` virtual member fn

Called by `Latte_Obj::visit()` when the `Latte_Obj` is actually a `Latte_Operator`. The default implementation does nothing.

`void` **visit_param** (`Latte_Param` &*param*) `Latte_Visitor` virtual member fn

Called by `Latte_Obj::visit()` when the `Latte_Obj` is actually a `Latte_Param`. The default implementation does nothing.

`void` **visit_str** (`Latte_Str` &*str*) `Latte_Visitor` virtual member fn

Called by `Latte_Obj::visit()` when the `Latte_Obj` is actually a `Latte_Str`. This function is "pure virtual" in `Latte_Visitor`; there is no default implementation, and subclasses are required to override it.

In `latte-html`, the `LatteHtml_HtmlVisitor` class supplies an implementation of `visit_str()` that works as follows:

- It calls `Latte_Visitor::suggest_wstate()` to determine the proper whitespace to emit before the string;
- It emits the suggested whitespace;
- If the whitespace included two or more newlines, and if processing is not presently inside a call to `{\pre ...}`, it emits the HTML paragraph tag '`<p>`';
- If processing is presently inside a call to `{\html ...}`, it emits the string literally; otherwise it emits each character of the string one at a time, converting '`<`', '`>`', '`&`', and '`&quot;`' to '`&lt;`', '`&gt;`', '`&amp;`', and '`&quot;`', respectively.

`void` **visit_varref** (`Latte_VarRef` *&varref*) — `Latte_Visitor` virtual member fn

Called by `Latte_Obj::visit()` when the `Latte_Obj` is actually a `Latte_VarRef`. The default implementation does nothing.

`void` **visit_wsnode** (`Latte_WsNode` *&wsnode*) — `Latte_Visitor` virtual member fn

Called by `Latte_Obj::visit()` when the `Latte_Obj` is actually a `Latte_WsNode`. The default implementation notes the whitespace contained in *wsnode* for possible use in subsequent calls to `suggest_wstate()`; then it calls `Latte_Obj::visit()` on the nested object within *wsnode*.

6 Whitespace and file locations

7 Shared strings

8 Miscellaneous details

8.1 Reference counting

8.2 Debug log

8.3 Restorer

A `Restorer` is an object that holds a reference to some variable, and a value for that variable. It assigns the value to the variable when it goes out of scope.

Its name comes from the fact that it's usually used to restore the value of a variable that has been temporarily changed for the duration of some function. For example, here's how `Latte_Obj::eval()` keeps track of its recursion depth:

```
static unsigned int depth = 0;
Restorer<unsigned int> depth_restorer(depth);

++depth;

...
val = do_eval(activation);
...
```

At the end of the function, `depth` is restored to the value it had before the `++depth` statement. This is more reliable than

```
++depth;

...
val = do_eval(activation);
...

--depth;
```

since `do_eval()` may throw an exception which could cause the `--depth` to be skipped. A `Restorer`, on the other hand, will restore the old value even when the scope ends because of an exception.

Restorer<T> (`T &`*var*) Constructor

Constructs a new `Restorer` on a variable of type `T`. The current value of *var* is recorded. When this `Restorer` is destructed, that value is assigned back to *var*.

Restorer<T> (`T &`*var*`, const T &`*now*`, const T &`*later*) Constructor

Alternative constructor that immediately assigns to *var* the value *now*, and that assigns *later* to *var* when the `Restorer` is destructed.

9 Writing new Latte applications

Index

Table of Contents