Latte

The Language for Transforming Text

Bob Glickstein

1 Introduction

Latte, the Language for Transforming Text, is a very simple and powerful language for including *markup* in text documents. Markup refers to information in the text that isn't strictly part of the text but describes it—for example, to specify that a certain word should appear in boldface, or that a certain phrase is to be rendered as a chapter heading.

Many languages for text markup already exist. The best-known one is HTML (Hypertext Markup Language), the language of World-Wide Web documents. Other well-known markup languages are SGML, TEX, and troff.

Why create a new markup language? Because the existing languages lack generality. HTML can't easily be used for anything except web documents. TEX can't easily be used for anything except printed books, articles, business letters, and the like.

Latte, on the other hand, defines a very simple syntax that is suitable for every task requiring text markup. Latte also provides a software library that makes it easy for programmers to create *translators* from Latte into other languages. One such translator, `latte-html`, comes with Latte and can convert Latte documents into HTML. Another, `latte-text`, produces a plain text version of the very same source document. Many other Latte translators are planned.

HTML authors generally find it easier to write in Latte than in HTML (see Section 3.6 [Latte vs. HTML], page 10).

Latte documents can therefore be used for everything HTML is used for *and* everything TEX is used for *and* everything troff is used for, etc. In fact, a single Latte document can be run through different translators to produce HTML, TEX, and plain text versions.

Latte is dedicated to my sister, the amazing Suzanne Glickstein.

2 Latte examples

Here are some examples illustrating the use of Latte to write World Wide Web documents.

```
{\head {\title Our document}}
{\body
 {\h1 Our document}

 Here is our first WWW document produced with
 {\a \href=http://www.latte.org/ Latte}.

 We can't wait to get started on our second document!}
```

This short Latte document produces the following HTML when run through `latte-html -f`:

```
<head><title>Our document</title></head>
<body><p><h1>Our document</h1>

 <p>Here is our first WWW document produced with
 <a href="http://www.latte.org/">Latte</a>.

 <p>We can't wait to get started on our second document!</body>
```

Observe that Latte uses curly braces (‘`{`’ and ‘`}`’) to group text, and a backslash (‘`\`’) to introduce markup keywords.

This example only uses Latte keywords that are exact analogs of HTML tags: `head`, `title`, `body`, `h1`, `a`, and the attribute `href`. It's possible to write Latte documents that contain nothing but HTML analogs, but the real power of Latte is revealed when its own features are combined with those of the target language.

For instance, note that the document title of the previous example, `Our document`, appears twice in the Latte and HTML versions—once inside the `title` tag and once inside the `h1` tag. It shouldn't be necessary to write the document's title twice, though. It should suffice to write it once and by some mechanism have it appear in the two places it's needed. Also, writing it twice is an invitation to error: if the document title ever changes, it's possible to update it in one place and forget to update it in the other.

Here's a version of the first example that solves that problem:

```
{\def \doctitle {Our document}}
{\head {\title \doctitle}}
{\body
 {\h1 \doctitle}

 Here is our first WWW document produced with
 {\a \href=http://www.latte.org/ Latte}.

 We can't wait to get started on our second document!}
```

This version begins by using the Latte procedure `\def` to define a new *variable* named `\doctitle`. The value of `\doctitle` is `Our document`. This value is substituted in the two places that `\doctitle` is used when this file is processed with `latte-html`.

Now suppose the enthusiastic authors of this example decide that every WWW document they write will have the same text in the `title` section as in an `h1` section at the beginning of the document body. Rather than structure every Latte file along these lines:

```
{\def \doctitle ...}
{\head {\title \doctitle}}
{\body
 {\h1 \doctitle}

 ...}
```

they can define a *function* to do that work for them:

```
{\def {\document \doctitle \&content}
  {{\head {\title \doctitle}}
   {\body
     {\h1 \doctitle}}

     \content}}
```

and can now write each document in abbreviated fashion:

```
{\document {Our second document}
  This was as exciting to
  write as our first!}
```

3 Latte syntax

When a Latte translator program such as `latte-html` reads a Latte file, it considers the contents as a sequence of Latte *expressions*. An expression is one of the following things:

- A *word*—that is, any sequence of characters containing no whitespace, no backslash, and no curly braces;
- A *quoted string*—that is, any sequence of characters enclosed in a pair of `\" ... \"` delimiters.
- A *variable reference*—that is, a backslash followed by the name of a variable. Variable names in Latte must begin with a letter or an underscore, and may be followed by zero or more letters, underscores, or digits, plus the characters ‘`?`’, ‘`!`’, ‘`+`’, and ‘`-`’.
- A *group*, which is a pair of curly braces surrounding zero or more *subexpressions*.
- A *quoted expression*—that is, any Latte expression preceded by one of Latte's quote symbols: `\'`, ``\` ``, `\,`, and `\,@`.

These are all covered in more detail in the sections that follow. Other constructs that may appear in a Latte file are:

- Whitespace, naturally. Latte handles whitespace in a special way; see Section 3.5 [Whitespace], page 9.
- Comments. These begin with ‘`\;`’ and continue to the end of the line. Latte discards comments when reading Latte files.
- The “forget-whitespace” symbol, written ‘`\/`’. Wherever this appears, it has the effect of canceling all the whitespace Latte has read since the last expression.

3.1 Words and strings

Most Latte files consist mainly of “words.” Simply put, words are those sequences of characters that aren't categorized as any other kind of Latte construct. To be more precise, a Latte word is a sequence of characters that does not include whitespace, backslashes, or curly braces. However, any of those characters *can* be included in a word by *escaping* them with a backslash.

Here's an example:

```
In a Latte "word," you must use \\ to escape \\, \{ and \}.
```

This example contains these fourteen Latte words:

1. In
2. a
3. Latte
4. "word,"
5. you
6. must
7. use
8. \

9. to
10. escape
11. `\`,
12. `{`
13. and
14. `}`.

In Latte, whitespace is not normally part of a word (it's handled specially; see Section 3.5 [Whitespace], page 9). Sometimes, however, it's desirable to force Latte to consider a single word to include some whitespace, especially when you wish to affix whitespace to the beginning or end of a word. This is best accomplished by using a quoted string, which Latte treats exactly like a word. A quoted string begins with '`\"`' and ends with '`\"`'. Inside a quoted string, only backslash must be escaped (with a backslash); every other character, including '`"`', '`{`', and '`}`', can be written normally. Quoted strings are therefore also useful for text that contains a lot of curly braces, such as fragments of C or Perl programs.

Note that the only way to represent an empty word in Latte is with quoting: `\"\"`.

3.2 Variables

A Latte variable can hold a single Latte expression as its value. That value is substituted wherever the variable is used.

The expression

```
{\def \varname expression}
```

creates a new variable named *varname* whose value is *expression*.

The expression

```
{\set! \varname expression}
```

assigns the new value *expression* to *varname*, which must previously have been defined with `\def`.

The expression

```
{\let {{\varname1 expression1}
       {\varname2 expression2}
       ...}
   body ...}
```

assigns *expression1* to *varname1*, *expression2* to *varname2*, and so on, then executes *body* with those variables in effect. Outside of the `\let` expression, those variables do not exist.

3.2.1 Scope

It is possible to have a variable outside a `\let` with the same name as a variable inside a `\let`. Suppose the variable is `\foo`. Then the definition inside the `\let` *hides* or *shadows* the definition outside the `\let`, like so:

```
{\def \foo 7}

{\let {{\foo 12}}
```

```
       The value of foo is \foo}

     Now the value of foo is \foo
```

This example yields the following text:

> The value of foo is 12
>
> Now the value of foo is 7

The region in which a variable is visible is called its *scope*. Different computer languages have different *scoping rules*. Latte uses *lexical scope*, also sometimes called *static scope*. This means that you can always tell by looking at the Latte file exactly which variable is meant by each variable reference—it's always the one defined in the textually innermost enclosing scope.[1]

3.2.2 Function parameters

The parameters of Latte functions (see Section 3.3 [Groups], page 8) are variables whose scope is the body of the function for which they're defined. There are three kinds of function parameter in Latte:

positional parameter

This is the simplest kind of parameter. In the definition of the function, a positional parameter is written exactly like an ordinary variable. When the function is invoked, the first actual argument is assigned to the first positional parameter, the second actual argument is assigned to the second positional parameter, and so on.[2] Here's an example of a function with two positional parameters:

```
{\def {\function \param1 \param2}
  {My arguments are \param1 and \param2.}}

{\function red green}
  ⇒ My arguments are red and green.
```

If fewer arguments are given than there are positional parameters, then the excess parameters get the default value `{}` (i.e., the empty group). If too many arguments are given and there is no "rest parameter" (see below), the excess arguments are silently ignored.

named parameter

When a function is invoked, a named parameter only gets a value when the caller explicitly assigns it a value by name. In the parameter list of the function

[1] Under another popular set of scoping rules called *dynamic scope*, the variable meant by a particular variable reference can actually change from moment to moment, depending on differences in the order in which functions are executing.

[2] What's the difference between a "parameter" and an "argument"? The terms are usually used interchangeably, but technically speaking, "parameter" refers to the variable in the function definition, while "argument" is the value that gets passed in when the function is called. The value of the argument is assigned to the parameter.

definition, a named parameter is indicated by `\=`*param*. When the function is called, its named parameters are given values by writing `\`*param*`=`*value*. (There must be no whitespace between `\`*param* and the `=` that follows it.) In the body of a function, named parameters are referenced like ordinary variables.

Example:

```
{\def {\function \=param}
  {\if \param
    {My param is \param}}}

{\function \param={full of eels}}
  ⇒ My param is full of eels
```

Any named parameter not given a value gets the value `{}` by default.

Note that one difference between positional parameters and named parameters is that the caller of a function need not know the names of positional parameters in order to use them, but must know the names of named parameters in order to use those.

When a function is invoked, any named parameter assignments do not affect matching up arguments to *positional* parameters. Named parameter assignments may be given in any order, anywhere in the argument list, without changing the meaning of the function call.

rest parameter

Every function may have one rest parameter, which is denoted in the parameter list by `\&`*param*. The value of this parameter is a Latte group containing excess arguments that could not be assigned to positional parameters. Example:

```
{\def {\function \param1 \param2 \&rest}
  {The first two parameters are \param1 and \param2
   The remaining parameters are \rest}}

{\function a b c d e}
  ⇒ The first two parameters are a and b
    The remaining parameters are c d e
```

Latte makes extensive use of rest parameters in functions that are HTML analogs, allowing Latte to represent an arbitrary amount of enclosed text easily (see Section 5.2.2 [Balanced tag functions], page 26). For example, here is a simplified definition of `\a` (with some essential details removed for clarity), which emits an HTML anchor:

```
{\def {\a \=href \&rest}
  {<a {\if \href
        {href="\href"}} >
   \rest
   </a>}}
```

Now when `\a` is called as follows:

```
{\a \href=http://www.latte.org/
    The Language for Transforming Text}
```

the five excess arguments—`The`, `Language`, `for`, `Transforming`, and `Text`—are grouped together in `\rest` and emitted between the `<a>` and `</a>` tags, yielding

```
<a href="http://www.latte.org/">
The Language for Transforming Text
</a>
```

(One of the details removed for clarity is the precise whitespace arrangement in the output of this example, but we return to this example with greater precision in Section 3.5 [Whitespace], page 9.)

3.3 Groups

A group in Latte is an expression composed of a sequence of zero or more subexpressions surrounded by '`{`' and '`}`'.

Groups serve two purposes. The first is to gather multiple expressions into a single expression, for use in contexts where only a single expression is allowed. For example, only a single expression may be passed as the value when defining a variable with `\def`. If the value happens to be a single word, grouping isn't needed:

```
{\def \var Hello}
```

but for more complex values, grouping is needed:

```
{\def \var {Hello there}}
```

The other use of groups in Latte is for performing *function calls*. When the first subexpression of a group is a function (that is, when it is a variable reference whose *value* is a function, or when it is some other Latte expression yielding a function), then that function is called, passing the remaining subexpressions to it as arguments. (The rules for how arguments are interpreted by functions are described in Section 3.2.2 [Function parameters], page 6.) The entire group is then replaced with the value produced by the function call.

Of special interest is the variant of `\def` that looks like this:

```
{\def {\var ...}
  body ...}
```

When the first argument to `\def` is a group rather than a variable reference, then it is interpreted as a function definition. For details on using `\def`, see Chapter 4 [Latte builtins], page 13.

3.4 Quoted expressions

A quoted expression is a Latte expression preceded by a quote symbol, of which there are four kinds described below.

Quoting expressions is for preventing them from being evaluated, or exerting control over *when* they're evaluated. In Latte this is generally necessary only when writing macros (see Section 4.3 [Functions on functions], page 15).

Here are the four kinds of quoted expressions:

`\'`*expr*

This is an *ordinary quote* that, when evaluated, yields *expr* exactly as written. Example:

```
\'{\add 3 4}
  ⇒ {\add 3 4}
```

\`*expr*

This is called *quasiquote* and works just like the ordinary quote, except that individual subexpressions of *expr* may be *unquoted*.

The result of evaluating a quasiquoted expression is that expression itself, except with any unquoted subexpressions replaced by their values. (See below for example.)

\,*expr*

This is an *unquote*. When it appears inside a quasiquoted expression, it causes *expr* to be evaluated. Example:

```
\`{\add \,{\multiply 3 4} 5}
  ⇒ {\add 12 5}
```

\,@*expr*

This is called *unquote-splicing*, or *splicing unquote*. It works just like unquote, except that if the result of evaluating *expr* is a group, then the result of \,@*expr* is *the individual elements of that group* rather than *the group containing the elements*. This is used for "splicing" the elements of a list into the containing list at the same nesting level. Example:

```
\`{a b \,@{\group c d e} f}
  ⇒ {a b c d e f}
```

See the documentation for `\macro` for an example using the different forms of quote.

3.5 Whitespace

As mentioned earlier, a Latte translator divides a Latte file into expressions. Whitespace is not considered an expression. Instead, each expression gets associated with *the whitespace that precedes it in the Latte file*. For example, when given the following input:

```
{\document {Our second document}
  This was as exciting to
  write as our first!}
```

Latte divides it up as follows:

- A group preceded by no whitespace, consisting of:
  - The variable reference `\document` preceded by no whitespace;
  - A group preceded by a single space, consisting of:
    - The word `Our` preceded by no whitespace;
    - The word `second` preceded by a single space;
    - The word `document` preceded by a single space
  - The word `This` preceded by one newline and two spaces;
  - The word `was` preceded by one space;

- The word `as` preceded by one space;
- The word `exciting` preceded by one space;
- The word `to` preceded by one space;
- The word `write` preceded by one newline and two spaces;
- The word `as` preceded by one space;
- The word `our` preceded by one space;
- The word `first!` preceded by one space.

The whitespace attached to an expression follows the expression wherever it goes (e.g., whether the expression is assigned to variables, passed to functions, etc.). However, that may not be the whitespace used when and if the expression is emitted as output. In general, when emitting output, expressions inherit the preceding-whitespace of the context in which they're used, not the context in which they're defined.

Here are some examples:

```
{\def \foo Hello}

"\foo"
  ⇒ "Hello"
```

In this example, even though `Hello` is preceded by a single space where it's assigned to `\foo`, the *use* of `\foo` has zero spaces preceding it, so the result contains no space between `"` and `Hello`.

```
{\def {\a \=href \&rest}
  {<a {\if \href
        {href="\href"}} >
   \rest
   </a>}}

{\a \href=foo bar}
  ⇒ <a href="foo" >
       bar
       </a>
```

Why, in the output, is there just one space between `a` and `href`? In the definition of `\a`, the subexpression `{href="\href"}` is preceded by a newline and several spaces. However, it's used inside an `\if` expression, which itself is preceded by only one space. When the `\if` expression is replaced by the `{href="\href"}` subexpression, the subexpression inherits its whitespace.

A word about the "forget-whitespace" symbol, `\/`. When this appears in a Latte file, all the whitespace that precedes it is canceled. This can be useful when you want the Latte source to include whitespace for readability, but you don't want that whitespace to appear in the output.

3.6 Latte vs. HTML

> HTML was never supposed to be something that you would see. . . it staggers me that people have actually put up with having to write HTML by hand.

—Tim Berners-Lee, inventor of HTML

Here are some shortcomings of HTML that are remedied by Latte.

Matching tags

In HTML, most markup tags come in beginning-ending pairs that must match and nest properly. For instance, `<b><i>text</i></b>` is how you'd put `text` in a bold-italic typeface. But if you write `<b><i>text</b></i>`, it's an error. Plus, all those paired tags create a lot of visual clutter, making HTML hard for humans to read. And most text editors can't offer much help in identifying and navigating matching sets of tags.

On the other hand, Latte uses curly braces to group text, which most text editors are already able to handle well. And Latte only requires beginning tags, not ending tags, reducing visual clutter and eliminating the potential for mismatches.

Paragraph tags

HTML requires `<p>` to appear at the beginning of each paragraph. Since most of us are accustomed to denoting paragraph breaks with just a blank line, it's easy to forget the `<p>`. When the `<p>` is remembered, it adds more clutter to the text.

When `latte-html` outputs one or more blank lines, it automatically includes a `<p>`. See Section 5.1 [Automatic HTML processing], page 25.

Metacharacters

HTML defines numerous *metacharacters*—characters that trigger special processing. These include '`<`', '`>`', '`&`', and '`"`'. Wherever you wish to include these characters in a document, you must use a named "character entity" instead: `&lt;`, `&gt;`, `&amp;`, and `&quot;`.

Not only are these characters bad choices for metacharacters in documents that are mostly textual (and may therefore be expected to contain several occurrences of '`"`' and '`&`' at least), but their character entity replacements are verbose and are not visually related to the characters they replace.

In Latte, the only metacharacters are '`\`', '`{`', and '`}`', and to use these characters literally, it is only necessary to escape them with a backslash. When `latte-html` encounters HTML metacharacters such as `&`, it automatically emits the corresponding HTML character entity.

Repeated text and constructs

HTML has no macro facility; no way to store text that is used repeatedly in a document, and no way to define shorthand for oft-used idioms. Consequently, HTML documents may contain redundant stretches of text (as we saw in the first example; see Chapter 2 [Latte examples], page 2) or complex, repeated, and difficult-to-diagnose layout directives.

Latte is a full-fledged programming language. Not only can repeated constructs be encapsulated in variables and functions, but these variables and functions can have descriptive names that indicate their purpose. Instead of dozens of nested `<table>`, `<tr>`, and `<td>` tags, for instance, each with dozens of attribute settings, one can lay out a complex page with something simple like:

```
{\layout \top-margin={...}
         \left-margin={...}
         ...main body...}
```

4 Latte builtins

This chapter is a reference guide to the built-in functions of Latte. It does not describe the HTML-related functions of `latte-html`. For those, see Section 5.2 [HTML functions], page 25.

Most of these functions are intrinsic to Latte. However, some are loaded at runtime from the library file '`standard.latte`'.

4.1 Control structures

if *test then-expr else-expr* Function

Evaluates *test*. If the result is true (see Section 4.5 [Boolean functions], page 19), evaluates *then-expr*, otherwise evaluates *else-expr*.

The result of the `\if` is the result of whichever of the *then-expr* or *else-expr* is invoked. If *else-expr* is omitted, it is taken to be `{}` by default.

while *test body . . .* Function

Evaluates *test*. If the result is true, evaluates the *body* subexpressions, then repeats.

The result is the *concatenation* of the results of each invocation of the *body* expressions. For example:

```
{\def \i 0}
{\while {\less? \i 3}
  i is \i
  {\set! \i {\add \i 1}}}
  ⇒ i is 0
    i is 1
    i is 2
```

foreach `\`*var group body . . .* Function

Evaluates *group*. Then, for each element in the result, sets `\`*var* to that value and evaluates the *body* subexpressions.

4.2 Functions on variables

def `\`*var value* Function

Creates a new variable `\`*var* in the innermost scope (see Section 3.2.1 [Scope], page 5), evaluates *value*, and assigns the result to the new variable.

def `{\`*var params . . .*`}` *body . . .* Function

Creates a new function whose parameters are given by *params* and whose body consists of *body*; assigns the function to the new variable `\`*var* in the innermost scope.

The parameter list *params* consists of zero or more positional parameters and named parameters, plus one optional rest parameter. See Section 3.2.2 [Function parameters], page 6.

When the function is invoked, the *body* expressions are evaluated one at a time in sequence. The return value of the function is the value of the last *body* expression.

> **This behavior differs from that of Latte versions prior to 2.0!**
> In those versions, user-defined functions constructed an implicit group containing the values of all *body* expressions. That behavior introduced an ambiguity in the case of functions that sometimes returned a group and sometimes returned some other kind of value. The new behavior more closely matches that of other related programming languages (see Appendix A [Pedigree], page 37).
>
> To ensure that a function written for the old Latte works under the current Latte, it should generally suffice to convert this:
>
> ```
> {\def {\func args ...}
>   body1 body2 ...}
> ```
>
> into this:
>
> ```
> {\def {\func args ...}
>   {body1 body2 ...}}
> ```
>
> To aid in this transition, a new kind of error is detected by Latte. Invoking a user-defined function that contains subexpressions that have no side effects and don't participate in the return value signals a "Useless subexpression" error.

Note that `{\def {\var params ...} body ...}` is exactly equivalent to

```
{\def \var
      {\lambda {params ...}
        body ...}}
```

The `\lambda` function is described in Section 4.3 [Functions on functions], page 15.

defmacro `{\var params ...}` *body ...* Function

Like `{\def {“var params ...} body ...`, but creates a macro instead of a function. See `\macro` (see Section 4.3 [Functions on functions], page 15).

let `{{\var value} ...}` *body ...* Function

Assigns to each \\var the result of evaluating each *value*, then evaluates *body* in the scope of those variables. Note that every *value* is evaluated before any of the \vars is assigned, so the *values* cannot refer to the \vars. In other words, this is an error:

```
{\let {{\list {x y z}}
       {\first {\front \list}}}
  ...}
```

To get the desired effect, you can do this:

```
{\let {{\list {x y z}}}
  {\let {{\first {\front \list}}}
    ...}}
```

or this:

```
{\let {{\list {x y z}}
       {\first {}}}
  {\set! \first {\front \list}}
  ...}
```

The value of the `\let` expression is the result of evaluating the last *body* expression.

> **This behavior differs from that of Latte versions prior to 2.0!**
> In those versions, `\let` expressions constructed an implicit group
> containing the values of all *body* expressions. See `\def`
> for a discussion of this change.
>
> To ensure that a `\let` expression written for the old Latte works
> under the current Latte, it should generally suffice to convert this:
>
> ```
> {\let {{“var val} ...}
>   body1 body2 ...}
> ```
>
> into this:
>
> ```
> {\let {{“var val} ...}
>   {body1 body2 ...}}
> ```
>
> To aid in this transition, a new kind of error is detected by Latte.
> Invoking a `\let` expression that contains subexpressions that have
> no side effects and don’t participate in the return value signals a
> “Useless subexpression” error.

Note that

```
{\let {{“var val} ...}
  body ...}
```

is equivalent to

```
{{\lambda {“var ...} body ...} val ...}
```

set! \var value Function

Assigns to existing variable \var the result of evaluating *value*.

4.3 Functions on functions

funcall *function args ...* Function

Evaluates *function*; the result must be a Latte function. Then evaluates all of the *args*. The function yielded by *function* is then invoked, with *args* passed in as its arguments.

```
{\funcall \add 2 3 4} ⇒ 9
```

apply *function args . . . last-arg* Function

Exactly like `\funcall`, except that if the value of *last-arg* is a group, then *its elements*, not the group, are passed individually to *function*.

```
{\apply \add 2 3 4} ⇒ 9

{\let {{\numbers {2 3 4}}}
  {\apply \add 6 \numbers}} ⇒ 15
```

lambda `{`*params . . .*`}` *body . . .* Function

Creates a Latte function with parameter list given by *params* and body given by *body*.

A full discussion of user-defined functions in Latte appears in the description for `\def` (see Section 4.2 [Functions on variables], page 13).

macro `{`*params . . .*`}` *body . . .* Function

Like `\lambda`, this produces a new Latte function with the given parameter list and body; but the function produced is a special kind called a *macro*. When the macro is evaluated by passing it some argument:

1. the arguments are *not evaluated*;
2. the macro body is evaluated using the non-evaluated argument values;
3. the result of *that* is then evaluated.

For example, here's how to use a macro to define a function called `\unless`.

```
{\def \unless
  {\macro {\test \&body}
   \`{\if {\not \,\test} \,\body}}}
```

The idea is for `{\unless` *test body* `...}` to evaluate *body* only when *test* is false. Here's how the macro definition works when `\unless` is called like this:

```
{\unless {\zero? \x}
  {\set! \x {\subtract \x 1}}}
```

1. The argument value `{\zero? \x}` is assigned to the macro parameter `\test`, and the remaining arguments (in this example, the sole body expression) are gathered into a group and assigned to `\body` (so `\body` equals `{{\set! \x {\subtract \x 1}}}`).

2. The body of the macro is evaluated. This is a quasiquoted expression (see Section 3.4 [Quoted expressions], page 8), so first, nested unquoted expressions are evaluated:

   1. `\,\test ⇒ {\zero? \x}`

   2. `\,\body ⇒ {{\set! \x {\subtract \x 1}}}}`

   and then the quasiquoted expression is returned with the unquoted subexpressions replaced by their values:

   ```
   {\if {\not {\zero? \x}}
        {{\set! \x {\subtract \x 1}}}}}
   ```

3. The resulting expression is evaluated normally.

compose *f1 f2* Function

Produces a new function of one argument that yields `{`*f1* `{`*f2 x*`}}` (where *x* is the argument). Naturally, *f1* and *f2* must both be functions of one argument.

This is used in the definition of `\cadr` et al. (see Section 4.4 [Group functions], page 17):

```
{\def \cadr {\compose \car \cdr}}
```

lmap *function group* Function

Evaluates *function*, which must yield a function, and *group*, which must yield a group. Then applies *function* to each element of *group* in turn, yielding a group containing the result of each function call.

Example:

```
{\def {\add1 \x} {\add \x 1}}
{\lmap \add1 {3 4 5}} ⇒ 4 5 6
```

4.4 Group functions

append *expr . . .* Function

Each *expr* is evaluated, and a group of expressions is constructed as follows: if *expr*'s value is a group, its elements are individually added to the result; otherwise *expr*'s value is added as a single element.

back *group* Function

Yields the last element of the given *group*.

car *group* Function

Yields the first element from the given *group*. Synonym for `\front`.

cdr *group* Function

Yields the elements of *group* minus the first element.

Example:

```
{\cdr {a b c}} ⇒ b c
```

caar *expr* Function
cadr *expr* Function
cdar *expr* Function
cddr *expr* Function

Produces `{\car {\car` *expr*`}}`, `{\car {\cdr` *expr*`}}`, `{\cdr {\car` *expr*`}}`, `{\cdr {\cdr` *expr*`}}`, respectively.

These functions are defined in '`standard.latte`' using `\compose` (see Section 4.3 [Functions on functions], page 15).

cons *expr group* Function

Constructs a new group consisting of *expr* followed by the elements of *group*. Synonym for `\push-front`.

empty? *group* Function

Yields a true value if *group* is empty, false otherwise. See Section 4.5 [Boolean functions], page 19. Equivalent to

```
{\equal? 0 {\length group}}
```

front *group* Function

Yields the first element from the given *group*. Synonym for `\car`.

group *expr . . .* Function

Constructs a group consisting of all the given *expr*s.

length *group* Function

Returns the number of elements in *group*. Can also be used on text strings; see Section 4.6 [Text functions], page 20.

member? *expr group* Function

Returns a true value if *expr* is `\equal?` to any member of *group*, false otherwise.

nth *n group* Function

Returns the *n*th element of *group*, counting from 0. Can also be used on text strings.

push-back *expr group* Function

Constructs a new group consisting of the elements of *group* followed by *expr*. Synonym for `\snoc`.

push-front *expr group* Function

Constructs a new group consisting of *expr* followed by the elements of *group*. Synonym for `\cons`.

rdc *group* Function

Yields the elements of *group* minus the last element.

Example:

```
{\rdc {a b c}} ⇒ a b
```

reverse *group* Function

Reverses the elements of *group*. Does not reverse the elements of nested groups.

Examples:

```
{\reverse {a b c}} ⇒ c b a

{\reverse {a {b c} d}} ⇒ d b c a
```

snoc *expr group* Function

Constructs a new group consisting of the elements of *group* followed by *expr*. Synonym for `\push-back`.

subseq *group from to* Function

Constructs a new group consisting of the elements of *group* beginning at *from* and ending before *to*. Both *from* and *to* count from 0. If either is negative, positions are counted backward from the end of *group* rather than forward from the front. If *to* is omitted, the new group contains the elements from *from* through the end of *group*.

Examples:

```
{\subseq {a b c d e} 1 3} ⇒ b c

{\subseq {a b c d e} -2} ⇒ d e
```

4.5 Boolean functions

Boolean values are truth and falsehood. In Latte, every value is considered “true” (for purposes of the test clauses in `\if` and `\while`) except for the empty group, `{}`, which is false. Note that the value `0` (zero), which is false in languages such as C and Perl, is true in Latte.

and *expr ...* Function

Evaluates each *expr* in turn until one yields falsehood. If no expression yields falsehood, returns the value of the last one. Otherwise returns `{}`.

not *expr* Function

Negates the truth value of *expr*. If *expr* is true, yields `{}`. If *expr* is false, yields the Latte “truth object.” (The “truth object” is a boolean value with no displayable representation that is only used for this purpose.)

or *expr ...* Function

Evaluates each *expr* in turn until one yields truth, then returns that value. If no expression is true, returns `{}`.

4.6 Text functions

concat *string . . .* Function

Constructs a new string by concatenating all the *string* arguments. Synonym for `\string-append`.

downcase *string . . .* Function

Constructs a group containing all the *string* values converted to lower case.

explode *string . . .* Function

Constructs a group whose elements are the individual characters of the given *string* values.

length *string* Function

Returns the number of characters in *string*. Can also be used on groups; see Section 4.4 [Group functions], page 17.

nth *n string* Function

Returns the *n*th character of *string*, counting from 0. Can also be used on groups.

string-append *string . . .* Function

Constructs a new string by concatenating all the *string* arguments. Synonym for `\concat`.

string-ge? *string . . .* Function

Returns truth if each *string* is greater than or equal to the ones following it ("monotonically non-increasing"), else returns falsehood. Synonym for `\string-greater-equal?`.

Example:

```
{\string-ge? three three ten ten seven one}
```

is true.

string-greater-equal? *string . . .* Function

Returns truth if each *string* is greater than or equal to the ones following it ("monotonically non-increasing"), else returns falsehood. Synonym for `\string-ge?`.

string-greater? *string . . .* Function

Returns truth if each *string* is strictly greater than the ones following it ("monotonically decreasing"), else returns falsehood. Synonym for `\string-gt?`.

string-gt? *string . . .* Function

Returns truth if each *string* is strictly greater than the ones following it ("monotonically decreasing"), else returns falsehood. Synonym for `\string-greater?`.

Example:

```
{\string-gt? three ten seven one}
```

is true.

string-le? *string ...* Function

Returns truth if each *string* is less than or equal to the ones following it ("monotonically non-decreasing"), else returns falsehood. Synonym for `\string-less-equal?`.

Example:

```
{\string-le? one seven ten ten three three}
```

is true.

string-less-equal? *string ...* Function

Returns truth if each *string* is less than or equal to the ones following it ("monotonically non-decreasing"), else returns falsehood. Synonym for `\string-le?`.

string-less? *string ...* Function

Returns truth if each *string* is strictly less than the ones following it ("monotonically increasing"), else returns falsehood. Synonym for `\string-lt?`.

string-lt? *string ...* Function

Returns truth if each *string* is strictly less than the ones following it ("monotonically increasing"), else returns falsehood.

Example:

```
{\string-lt? one seven ten three}
```

is true.

substr *string from to* Function

Constructs a new string consisting of the characters of *string* beginning at *from* and ending before *to*. Both *from* and *to* count from 0. If either is negative, positions are counted backward from the end of *string* rather than forward from the front. If *to* is omitted, the new string contains the characters from *from* through the end of *string*.

Examples:

```
{\substr abcde 1 3} ⇒ bc

{\substr abcde -2} ⇒ de
```

upcase *string ...* Function

Constructs a group containing all the *string* values converted to upper case.

4.7 Arithmetical functions

When Latte is installed, it can be configured for integer arithmetic only or for integer and floating-point arithmetic. If integer arithmetic only, then a number is any Latte string consisting of one or more digits, with an optional leading plus or minus sign. If integer and floating-point, then a number is any integer (as described) optionally followed by a decimal point and zero or more digits.

add *number ...* Function

Adds all the given *number*s.

ceil *number* Function

Rounds *number* up to the next integer. If Latte is configured for integer-only arithmetic, *number* is returned unchanged.

divide *number . . .* Function

Divides the first *number* by each successive argument. An attempt to divide by zero triggers an error.

floor *number* Function

Rounds *number* down to the next integer. If Latte is configured for integer-only arithmetic, *number* is returned unchanged.

ge? *number . . .* Function

Returns truth if each *number* is greater than or equal to the ones following it ("monotonically non-increasing"), else returns falsehood. Synonym for `\greater-equal?`.

Example:

```
{\ge? 10 10 7 3 3 1}
```

is true.

greater-equal? *number . . .* Function

Returns truth if each *number* is greater than or equal to the ones following it ("monotonically non-increasing"), else returns falsehood. Synonym for `\ge?`.

greater? *number . . .* Function

Returns truth if each *number* is strictly greater than the ones following it ("monotonically decreasing"), else returns falsehood. Synonym for `\gt?`.

gt? *number . . .* Function

Returns truth if each *number* is strictly greater than the ones following it ("monotonically decreasing"), else returns falsehood. Synonym for `\greater?`.

Example:

```
{\gt? 10 7 3 1}
```

is true.

le? *number . . .* Function

Returns truth if each *number* is less than or equal to the ones following it ("monotonically non-decreasing"), else returns falsehood. Synonym for `\less-equal?`.

Example:

```
{\le? 1 3 3 7 10 10}
```

is true.

less-equal? *number . . .* Function

Returns truth if each *number* is less than or equal to the ones following it ("monotonically non-decreasing"), else returns falsehood. Synonym for `\le?`.

less? *number* . . . Function

Returns truth if each *number* is strictly less than the ones following it ("monotonically increasing"), else returns falsehood. Synonym for `\lt?`.

lt? *number* . . . Function

Returns truth if each *number* is strictly less than the ones following it ("monotonically increasing"), else returns falsehood. Synonym for `\less?`.

Example:

```
{\lt? 1 3 7 10}
```

is true.

modulo *a b* Function

Returns *a* modulo *b*.

multiply *number* . . . Function

Multiplies all the given *number*s.

random *n* Function

Returns a random integer greater than or equal to 0 and less than *n*.

subtract *number* . . . Function

Subtracts from the first *number* each successive argument. If only one *number* is given, negates the number.

zero? *number* Function

Returns truth if *number* is 0, falsehood otherwise. Equivalent to:

```
{\equal? 0 number}
```

4.8 File functions

file-contents *filename* Function

Returns, as a Latte string, the contents of the file named by *filename*.

process-output *program args* . . . Function

Runs *program* with arguments *args*, returning the output of the program as a Latte string.

load-file *filename* Function

Evaluates the contents of the Latte file *filename*. Textual results are discarded, but variable and function definitions and other side effects are retained.

load-library *filename* Function

Like `\load-file`, but searches for *filename* using Latte's library-search algorithm. For each directory in the Latte search path, tries to load '*filename*' from that directory, or if that fails, '*filename*`.latte`'.

The default Latte search path is set at installation time, usually to the directories '`/usr/local/share/latte`' and '`.`'. It can be changed by setting the environment variable `LATTE_PATH` to a colon-separated list of directories.

include *filename* Function

This works like `\load-file` but the text of *filename* is not discarded. This can therefore be used to assemble an aggregate Latte document out of smaller pieces.

4.9 Other functions

equal? *expr . . .* Function

Returns truth if all the *expr*s are equal, falsehood otherwise.

Equality is defined recursively on groups: two groups are `\equal?` if and only if each of their corresponding subexpressions are `\equal?`.

getenv *name* Function

Returns the value of the environment variable named *name*. If no such environment variable exists, returns falsehood.

error *text . . .* Function

Exits from Latte with an error message given by *text* plus the file location where `\error` was called.

group? *expr* Function

Returns truth if the given *expr* is a group, falsehood otherwise.

operator? *expr* Function

Returns truth if the given *expr* is an operator (i.e., function), falsehood otherwise.

string? *expr* Function

Returns truth if the given *expr* is a string, falsehood otherwise.

warn *text . . .* Function

Emits a warning message given by *text* plus the file location where `\warn` was called.

4.10 Predefined variables

__latte-version__ Variable

This variable is predefined to contain the version number of `latte-html`.

5 Writing HTML with Latte

Latte was originally created to provide a saner alternative to HTML. Although it has expanded in purpose, simplifying the production of HTML documents remains its primary use, at least for now. This chapter discusses how. An understanding of HTML is assumed.

5.1 Automatic HTML processing

As `latte-html` evaluates top-level Latte expressions, it performs two kinds of automatic HTML processing on the result before sending it to the output:

Automatic `<p>`-tag generation
: Wherever one or more blank lines appear, any subsequent text is preceded with a `<p>` tag.

Automatic character-entity translation
: Wherever an HTML metacharacter is encountered, its character-entity code is substituted; e.g., `&lt;` and `&gt;` for '`<`' and '`>`'.

These processing steps can be controlled using the `\_pre` and `\html` functions; see Section 5.2 [HTML functions], page 25.

5.2 HTML functions

This section describes the HTML functions available in `latte-html`. Most are defined in '`html.latte`', but some are intrinsic to `latte-html`.

_pre *text ...* Function

This is a subroutine of the HTML tag function `\pre` (which is used for including preformatted text). Any text enclosed in `{\_pre ...}` is not subject to automatic `<p>`-tag generation. So whereas `latte-html` would normally turn this:

```
First paragraph.

Second paragraph.
```

into this:

```
First paragraph.

<p>Second paragraph.
```

it turns this:

```
{\_pre First paragraph.

Second paragraph.}
```

into this:

```
First paragraph.

Second paragraph.
```

Note that `\_pre` does *not* turn off automatic character-entity translation (see `\html` below), though it is possible to use `\_pre` in combination with `\html` to achieve that effect.

html *text ...* Function

Text enclosed in `{\html ...}` is not subject to automatic HTML character-entity translation. In other words, where `latte-html` would normally translate this:

```
"Penn & Teller"
```

into this:

```
&quot;Penn &amp; Teller&quot;
```

it turns this:

```
{\html "Penn & Teller"}
```

into this:

```
"Penn & Teller"
```

This is of course essential to implementing all of the HTML-tag Latte functions, since they must be able to emit HTML metacharacters (particularly '`<`' and '`>`') literally.

Note that `\html` does *not* turn off automatic `<p>`-tag generation, though it is possible to use `\html` in combination with `\_pre` to achieve that effect.

All remaining HTML functions fall into three categories: character-entity functions; balanced tag functions; and non-balanced tag functions.

5.2.1 Character entity functions

For each named HTML character entity `&`*foo*`;`, '`html.latte`' defines a function of zero arguments named `\c-`*foo*. So, for example, to emit a copyright symbol, write `{\c-copy}` (which produces `&copy;`).

The `\c-`*foo* functions are implemented in terms of `\ch`. Write `{\ch` *name*`}` to emit `&`*name*`;`.

There is also `\chx`, which takes a two-digit hexadecimal number as an argument. It produces `&#`*num*`;`.

5.2.2 Balanced tag functions

"Balanced tag functions" are Latte functions that produce a balanced pair of HTML tags surrounding some text, with optional HTML attributes in the opening tag. For example, `\b` is a balanced tag function because

```
{\b some text}
```

produces

```
<b>some text</b>
```

There is a balanced tag function corresponding to every HTML tag defined in the HTML 4.0 "Transitional" standard (which includes deprecated forms). The Latte names of these functions are always all-lowercase.

Every tag function has optional named parameters (see Section 3.2.2 [Function parameters], page 6) corresponding to the attributes permitted for that tag. For instance, `\a` has named parameters `\href` and `\name`, among others. These attribute names are always all-lowercase too.

When you specify a value for an HTML attribute, the value is automatically surrounded with double quotes. For example,

```
{\a \href=http://www.latte.org/ The Latte language}
```

produces this:

```
<a href="http://www.latte.org/">The Latte language</a>
```

Some tags have boolean attributes that do not take values. For instance, in

```
<textarea readonly>Some text</textarea>
```

`readonly` is a boolean attribute. The Latte way to include this attribute is to write `\readonly=1`. The `1` is ignored; any true value will do. A false value will cause the attribute not to appear in the tag.

Every balanced tag function in `latte-html` accepts an arbitrary amount of text (as its rest parameter; see Section 3.2.2 [Function parameters], page 6) to be enclosed by the paired beginning-ending HTML tags.

5.2.3 Non-balanced tag functions

Non-balanced tags are those that are not paired with a closing tag and do not enclose text; for example,

```
<img src="foo.gif" alt="picture of foo">
```

(There is no `</img>` tag.)

Non-balanced tag functions in `latte-html` work exactly like balanced tag functions do, except that they only accept various named parameters as arguments (corresponding to their HTML attributes) and do not accept an arbitrary amount of text to enclose.

5.3 Nonstandard HTML

It is possible to write nonstandard HTML tags and attributes even though '`html.latte`' doesn't predefine tag functions or named parameters for them.

Nonstandard tags

To write a nonstandard balanced tag, use `\_bal-tag`. To write a nonstandard non-balanced tag, use `\_tag`. These functions are used to define all the HTML tag functions in '`html.latte`'.

_bal-tag *name attrs bools deprs nonstandard* \=*depr* \&rest Function

Emits a balanced pair of HTML tags named *name*. The tag's attributes and their values are given by *attrs*, which is a group of the form

{*attribute value attribute value* ...}

An attribute in *attrs* with a false *value* does not appear in the output.

The tag's boolean attributes and their (boolean) values are given by *bools*, which has the same form as *attrs*.

The tag's deprecated attributes and their values are given by *deprs*, which also has the same form as *attrs*. Any attributes in *deprs* with true values will cause a warning to be emitted if `\strict-html4` is true (see Section 4.10 [Predefined variables], page 24).

Any nonstandard attributes and their values are listed in *nonstandard* and will also cause warnings to be emitted if `\strict-html4` is true.

The named parameter `\depr`, if set to true, indicates that this tag is deprecated and should generate a warning if used and `\strict-html4` is true.

The rest parameter accumulates text to appear between the opening and closing HTML tags.

Example:

```
{\_bal-tag whee
           {foo bar right wrong}
           {x 1 y {}}
           {}
           {}
           The text in the tags}
```

yields

```
<whee foo="bar" right="wrong" x>The text in the tags</whee>
```

Of greater utility would be defining a tag function named `\whee` along these lines:

```
{\def {\whee \=foo \=right \=x \=y \&rest}
  {\_bal-tag whee
             {foo \foo right \right}
             {x \x y \y}
             {}
             {}
             \rest}}
```

so that you can then write

```
{\whee \foo=bar \right=wrong \x=1 The text in the tags}
```

to get the same result as above.

_tag *name params bools deprs nonstandard* `\=`*depr* Function

This works exactly like `\_bal-tag` (see above) except that it does not have a rest parameter, does not enclose text, and does not emit an "ending tag."

Example:

```
{\_tag zoom
       {whoosh zing}
       {}
       {}
       {}}
  ⇒ <zoom whoosh="zing">
```

Nonstandard attributes

Including nonstandard attributes in a standard HTML tag is quite a bit easier. Every tag function in '`html.latte`' includes a named parameter called `\nonstandard` whose value is a list of attribute-value pairs to include in the tag. So, for example,

```
{\b \nonstandard={boldness high}
    Very bold!}
  ⇒ <b boldness="high">Very bold!</b>
```

even though `\b` does not have a `boldness` parameter.

Use of `\nonstandard` will trigger warnings if `\strict-html4` is true.

5.4 Predefined variables in latte-html

In addition to the variables predefined by all Latte translators (see Section 4.10 [Predefined variables], page 24), `latte-html` predefines the following additional variables:

__FILE__ Variable

The name of the file being processed.

strict-html4 Variable

This variable contains a true value if `latte-html` was invoked with the `--strict` option (see Section 5.5 [Invoking latte-html], page 29), otherwise it contains falsehood.

5.5 Invoking latte-html

Usage: `latte-html` *[options] [file]*

If *file* is not given or is `-`, the standard input is processed.

Options are:

`-v`
`--version`

Print the version number of `latte-html` and exit.

`-h`
`--help` Print a short help summary and exit.

`-l` *library*
`--load=`*library*

Load the Latte library file *library* before processing *file*. Any number of library files may be preloaded this way. Loading works as in `\load-library` (see Section 4.8 [File functions], page 23). Libraries requested with this option are loaded after the default libraries (see `-n`).

`-n`
`--no-default`

Do not load the default libraries. Normally, `latte-html` implicitly loads '`standard.latte`' (see Chapter 4 [Latte builtins], page 13) and '`html.latte`' (see Section 5.2 [HTML functions], page 25) on startup.

If you wish to suppress loading of just one default library file, use `-n` and also use `-l` to explicitly load the other library file, like so:

```
latte-html -n -l standard ...
```

`-o` *file*
`--output=`*file*
Place output in *file*. Without this option, `latte-html` places its output on the standard output.

`-f`
`--fragment`
Produce an HTML fragment rather than a complete HTML document. This causes `latte-html` to suppress the `<!DOCTYPE ...>` and `<html>` tags with which it normally surrounds its output. This can be useful when multiple Latte documents must be processed separately then combined to create a single HTML document.

`-s`
`--strict` Causes `latte-html` to emit warnings whenever HTML tags or attributes are used that have been deprecated in the HTML 4.0 standard.

The Latte variable `--strict` is set to true if this option is given, otherwise it's set to false.

`-L` *lang*
`--lang=`*lang*
Set the language code of the resulting HTML document to *lang*. This causes the `<html>` tag that is automatically emitted to include the `lang=`*lang* attribute.

`-d` *flags*
`--debug=`*flags*
Turn on Latte debugging; *flags* is a comma-separated list of flags chosen from the set `eval` and `mem`. When the `eval` debugging flag is turned on, each Latte evaluation and result is displayed, with its nesting level indicated with indentation. When the `mem` debugging flag is turned on, a summary of extant and total objects allocated is periodically displayed.

The output format of `--debug=eval` is designed to be used with the "selective display" feature of Emacs (q.v.), which is able to hide lines with greater than a given amount of indentation.

5.6 Makefile rules

It is convenient to delegate the work of invoking `latte-html` to `make`, especially when your project involves multiple Latte files that must be converted into multiple HTML files.

Here's a suffix rule you can add to your '`Makefile`':

```
.latte.html:
        -rm -f $@
        latte-html -o $@ $<
```

You may also find it convenient to list your HTML targets like so:

```
HTMLFILES = foo.html bar.html ...

all: $(HTMLFILES)
```

This, combined with the suffix rule above, will cause `make all` to create '`foo.html`' from '`foo.latte`', '`bar.html`' from '`bar.latte`', and so on.

Autoloading shared Latte definitions

It is typical to place common Latte function definitions into a separate file that is shared by all the other Latte files in a project. If you have such definitions in '`defns.latte`' (say), you may wish to change the suffix rule to

```
.latte.html:
        -rm -f $@
        latte-html --load=defns -o $@ $<
```

so that '`defns.latte`' is automatically loaded as each Latte file is processed. (The alternative is to begin each Latte file with a call to `{\load-library defns}`.)

Managing multi-author documents

Latte can aid in producing HTML documents from multiple sources, such as documents written by multiple authors, each of whom "owns" a section of the document. Each section can be its own Latte file, allowing all authors to edit their portions concurrently. The final document can be a Latte file that uses `\include` to include the subparts (see Section 5.2 [HTML functions], page 25), or it can be an HTML template that, when glued together with HTML fragments generated from the individual Latte files, forms a complete HTML file.

Here's an example of the first approach. Suppose Hank and Lorna are each responsible for one piece of '`article.html`', which is produced from the Latte file '`article.latte`'. Hank edits his portion in '`hank.latte`', and Lorna edits hers in '`lorna.html`'.

The file '`article.latte`' may look like this:

```
{\head {\title Our article}}
{\body
 {\h1 Thesis}

 {\include hank.latte}

 {\h1 Antithesis}

 {\include lorna.latte}}
```

and the final version can be produced simply by running `latte-html article.latte`. In this example, the files '`hank.latte`' and '`lorna.latte`' need not (indeed, should not) contain any calls to `\head` or `\body`.

The second approach, where the output of individual Latte files is pasted together into an HTML template, is slightly trickier, but not much. Imagine that '`article.html`' is to be produced from '`head.html`' and '`foot.html`', which presumably contain HTML boilerplate for beginning and ending files, with the middle of the document supplied by '`hank.latte`' and ‘lorna.latte’. The ‘Makefile’ rules for creating this document must generate HTML files from the Latte pieces, then concatenate all the HTML parts with the Unix command cat. Here’s how it can do that:

```
hank.html: hank.latte
        -rm -f hank.html
        latte-html --fragment -o hank.html hank.latte

lorna.html: lorna.latte
        -rm -f lorna.html
        latte-html --fragment -o lorna.html lorna.latte

article.html: head.html foot.html hank.html lorna.html
        -rm -f article.html
        cat head.html hank.html lorna.html foot.html > article.html
```

Note the use of the --fragment option to latte-html to suppress the automatic generation of an HTML preamble (containing a DOCTYPE declaration and commentary), an <html> tag, and a closing </html> tag.

5.7 Latte mode for Emacs

The Latte package comes with ‘latte.el’, which is an implementation of an Emacs editing mode for Latte files. It’s based on Emacs’s text mode and provides some consistent indentation rules plus syntax-based coloring of Latte language elements.

When Latte is installed, ‘latte.el’ (and a “byte-compiled” version of it, ‘latte.elc’) is installed the proper place for Emacs extensions. To use Latte mode while editing a Latte file, first ensure that the library is loaded by typing

M-x load-library ⟨RET⟩ *latte* ⟨RET⟩

then invoke latte-mode with

M-x latte-mode ⟨RET⟩

Since this is somewhat cumbersome, you may wish to have Emacs invoke Latte mode automatically when you edit files whose names end in ‘.latte’. To arrange this, add the following lines to your ‘.emacs’ file:

```
(autoload 'latte-mode "latte" "Latte mode" t)

(setq auto-mode-alist
      (cons '("\\.latte$" . latte-mode)
            auto-mode-alist))
```

6 The latte-text translator

Latte includes a program called `latte-text` that works very much like `latte-html` except that it produces plain text as output. The implementation is very rudimentary at present, but it's suitable for experimenting with. In future releases, `latte-text` will become more sophisticated.

In principle, a file that is written to be processed with `latte-html` can also be processed with `latte-text` to obtain a reasonable plain-text facsimile of the HTML version. The library file '`text.latte`' defines most of the same markup functions as does '`html.latte`' but gives them non-HTML definitions. For example, in `html.latte`, the function `\b` ("boldface") surrounds its arguments with `<b>` and `</b>`. In `text.latte`, the same function surrounds its arguments with *asterisks*.

7 Future directions

Here's what's planned for Latte beyond the current release.

Other translators and tools

Translators to languages other than HTML would increase the usefulness of Latte. One vision is for marked-up text to always be writable in Latte format, then translatable to whatever language is required for its final processing step (HTML for documents destined for the web, TEX for documents destined for the printer, ASCII for documents destined for plain-text displays, etc.). Whether that's actually a desirable goal is an open question, but it's certainly an interesting idea. At any rate, additional translators can only be a good thing.

There is a niche for tools that process Latte files without necessarily translating them. For instance, a Latte dependency-tracker can determine what files a particular one depends on via calls to `\file-contents`, `\include`, and so on. This would aid in writing accurate dependencies in Makefiles.

Higher-level libraries

The functions now available in `latte-html` provide a bare minimum of basic functionality plus HTML equivalence. They don't include any useful higher-level functions to assist with such things as page layout—you still have to write HTML-analogous code. Of course, Latte allows you to encapsulate that code as reusable functions, and indeed many such functions have been written by Latte users for specific purposes. Some of those functions should be cleaned up, documented, made more general, and assembled into a useful library.

Another set of high-level functions could abstract away the details of the target language. It should be possible to write a single Latte document that can produce both HTML and TEX, for example. But a Latte file containing calls to `\img` and `\h3` clearly has an HTML bias, whereas a Latte file containing calls to (hypothetical functions) `\setlength` and `\verbatim` clearly has a TEX bias. A set of common markup functions that translates well into all target languages could be just what the doctor ordered.

Richer string handling

Latte needs some Perl-like facilities for pattern matching, composition, and decomposition of strings.

Improved system interface

Latte should be able to perform file operations such as linking and unlinking files, testing access permissions, reading directories, and so on, as well as other system operations such as querying the current time, getting and setting environment variables, and so on.

Character set awareness

Presently, `latte-html` presumes the character set of the text in its input is ASCII or a superset thereof (such as ISO Latin-1). It should become possible to advise `latte-html` that the text is in some other character set. This would produce character set information in the generated HTML file, and would also affect which characters undergo automatic character-entity translation (see Section 5.1 [Automatic HTML processing], page 25), and what entities they're translated to.

Better debugging

The debugging output produced by `latte-html -d eval` is voluminous and useful only to the very dauntless. It should be possible to usefully restrict what output is seen, and that output should become more representative of the actions of the Latte engine.

Apache module

It should be possible to write a module for the Apache HTTP server that would allow it to serve Latte documents without their needing to be translated to HTML first; the translation would occur on the fly at document-serving time.

8 Further information

Further information about Latte can be found on the Latte home page at

`http://www.latte.org/`

To participate in Latte-related discussion, you may subscribe to the Latte mailing list by sending a request to

`latte-request@zanshin.com`

To reach the developers, send mail to `latte-dev@zanshin.com`. Use this address for reporting bugs. Please make sure your bug reports contain as much relevant information as possible, including:

- The version number of Latte (find out by running `latte-html --version`);
- Your operating system type and version number;
- The arguments you used to invoke `latte-html`;
- A concise description of the problem, including what you expected should happen and what actually happened;
- The smallest data sample you can devise that reliably demonstrates the problem, and the erroneous output, if any, that it produces.

Before reporting a bug, please be sure that the problem isn't a Latte usage error on your part. The Latte "Frequently Asked Questions" page can help you make this determination. It's at

`http://www.latte.org/faq.html`

Latte is *open source* software, licensed under the terms of the Zanshin Public License. A copy of the license comes with Latte in the file '`COPYING`'. The license can also be found at

`http://www.zanshin.com/ZPL.html`

Latte is a product of Zanshin, Inc. More about Zanshin can be found on the Zanshin home page at

`http://www.zanshin.com/`

Appendix A Pedigree

> The language designer's task is consolidation, not innovation.
>
> —C.A.R. Hoare (paraphrased)

Some users may recognize the influence of other languages in the design of Latte.

Latte borrows the choice of metacharacters ('`\`', '`{`', and '`}`') from TEX. One of TEX's drawbacks is that there actually are numerous other metacharacters, some of which are active in some contexts and not in others. This makes it tricky to write syntactically correct TEX, so Latte specifically rejects making any other characters "meta," even though some of TEX's metacharacters are frequently convenient (such as '`~`' for a non-breaking space).

Many of Latte's other features come from Scheme (a dialect of Lisp), including: **prefix notation**, in which function calls are written with the operator preceding the operands; **lexical scope**, described in section Section 3.2.1 [Scope], page 5, where the binding of a variable is fixed at lexical-analysis time (i.e., when the program is parsed rather than when it runs); **manifest types**, which means that "type" is a property of a value rather than of a variable; **quasiquoting**, which aids in writing macros; and **first-class functions**, meaning that function objects (created with `\lambda` and `\macro`; see Section 4.3 [Functions on functions], page 15) can be assigned to variables or passed to and returned from other functions.

Function and Variable Index

_

__FILE__ 29
__latte-version__ 24
_bal-tag 27
_pre 25
_tag 28

A

add 21
and 19
append 17
apply 16

B

back 17

C

caar 18
cadr 18
car 18
cdar 18
cddr 18
cdr 18
ceil 22
compose 17
concat 20
cons 18

D

def 13
defmacro 14
divide 22
downcase 20

E

empty? 18
equal? 24
error 24
explode 20

F

file-contents 23
floor 22
foreach 13
front 18
funcall 15

G

ge? 22
getenv 24
greater-equal? 22
greater? 22
group 18
group? 24
gt? 22

H

html 26

I

if 13
include 24

L

lambda 16
le? 22
length 18, 20
less-equal? 22
less? 23
let 14
lmap 17
load-file 23
load-library 23
lt? 23

M

macro 16
member? 18
modulo 23
multiply 23

N

not 19
nth 18, 20

O

operator? 24
or 19
ordinary quote 8

P

process-output 23
push-back 18

push-front 18

Q

quasiquote 9
quote 8

R

random 23
rdc 19
reverse 19

S

set! 15
snoc 19
splicing unquote 9
strict-html4 29
string-append 20
string-ge? 20
string-greater-equal? 20
string-greater? 20
string-gt? 20
string-le? 21
string-less-equal? 21
string-less? 21
string-lt? 21
string? 24
subseq 19
substr 21
subtract 23

U

unquote 9
unquote-splicing 9
upcase 21
useless 14

W

warn 24
while 13

Z

zero? 23

Table of Contents

1 Introduction 1

2 Latte examples 2

3 Latte syntax 4

3.1 Words and strings 4
3.2 Variables 5
3.2.1 Scope 5
3.2.2 Function parameters 6
3.3 Groups 8
3.4 Quoted expressions 8
3.5 Whitespace 9
3.6 Latte vs. HTML 10

4 Latte builtins 13

4.1 Control structures 13
4.2 Functions on variables 13
4.3 Functions on functions 15
4.4 Group functions 17
4.5 Boolean functions 19
4.6 Text functions 20
4.7 Arithmetical functions 21
4.8 File functions 23
4.9 Other functions 24
4.10 Predefined variables 24

5 Writing HTML with Latte 25

5.1 Automatic HTML processing 25
5.2 HTML functions 25
5.2.1 Character entity functions 26
5.2.2 Balanced tag functions 26
5.2.3 Non-balanced tag functions 27
5.3 Nonstandard HTML 27
5.4 Predefined variables in latte-html 29
5.5 Invoking latte-html 29
5.6 Makefile rules 30
5.7 Latte mode for Emacs 32

6 The latte-text translator 33

7 Future directions **34**

8 Further information **36**

Appendix A Pedigree **37**

Function and Variable Index **38**