Vesti Transpiler User Manual

Sungbae Jeong

February 16, 2026

Contents

1 Introduction

I used to create several documents using LaTeX (or plain TeX but TeX is quite cumbersome to write—especially when working with very complex tables or inserting images). Its markdown-like syntax is also not comfortable to use. For example, here is a simple LaTeX document:

```
% coprime is my custom class. See https://github.com/e0328eric/coprime.
\documentclass[tikz, geometry]{coprime}

\settitle{My First Document}{Sungbae Jeong}{}
\setgeometry{a4paper, margin = 2.5cm}

\begin{document}
\section{Foo}
Hello, World!
\begin{figure}[ht]
    \centering
    \begin{tikzpicture}
        \draw (0,0) -- (1,1);
    \end{tikzpicture}
```

```
\end{figure}

The code above is a figure using TikZ.

\end{document}
```

What annoys me most when using it is the '`\begin`' and '`\end`' blocks. Is there a way to write something much simpler? This question led me to start this project. Currently, the following code is generated into the LaTeXcode above (except comments) using vesti:

```
% coprime is my custom class. See https://github.com/e0328eric/coprime.
docclass coprime (tikz, geometry)

\settitle{My First Document}{Sungbae Jeong}{}
\setgeometry{a4paper, margin = 2.5cm}

startdoc

\section{Foo}
Hello, World!
useenv figure [ht] {
    \centering
    useenv tikzpicture {
        \draw (0,0) -- (1,1);
    }
}

The code above is a figure using TikZ.
```

Everywhere in this paper, the symbol ␣ denotes a "space" when one use Vesti.

2 Structure of Vesti File

Vesti is similar as LaTeX. Its structure consists with two parts: `preamble` and `main`. Preamble is the place where LaTeX documentclass, packages, and several settings are located. Main body is where actual documentation is located. Below figure is the simple Vesti documentation.

```
docclass article (10pt)
importpkg {
    geometry (a4paper, margin=2.2cm)
}
startdoc
Hello, Vesti!
```

We will see later, but the very difference with LaTeX is that Vesti has its own keywords (keywords are colored with purple). It makes the code readable and it is easier and faster to write the document. The keyword startdoc splits the preamble and the main part of the documentation similar with `\begin{document}` in LaTeX. However, Vesti does not have the analogous part of `\end{document}`, because almost every LaTeX document (99.999% I'm sure) does not have any code below `\end{document}`. For this reason, Vesti automatically ends document when EOF (End Of File) is found.

3 Keywords

Followings are reserved as keywords. In this document, every Vesti keyword has the form like `this`.

begenv	compty	cpfile	defenv
defun	docclass	endenv	importmod
importpkg	importves	startdoc	useenv

Table 1: Keywords in Vesti

3.1 docclass keyword

Keyword docclass is an analogous of \documentclass in LaTeX. If docclass is in the main paragraph, it acts just a normal word. In other words, docclass actives only in the preamble. The syntax of docclass is following:

docclass ␣ <class name> ␣ (<arguments>)

Here, arguments are separated by commas and embraced by (). Here are some examples.

- docclass ␣ article
- docclass ␣ article ␣ (10pt)
- docclass ␣ article ␣ (10pt, ␣ twocols)
- docclass ␣ article ␣ (10pt,twocols)

3.2 importpkg keywords

Keyword importpkg is an analogous of \usepackage in LaTeX. If importpkg is in the main paragraph, it acts just a normal word. In other words, importpkg actives only in the preamble.

importpkg has two different syntax. First one is same as docclass.

importpkg ␣ <pkg-name> ␣ (arguments)

Here, arguments are separated by commas and embraced by (). In the practical case, one should include several packages with options. importpkg also supports such case. We will look at an example instead of giving rigorous grammar.

```
importpkg {
    amsmath, amssymb, amsthm,
    geometry (a4paper, margin=2.2cm),
}
```

As one can see, inside of {}, several packages can be used together with thier options.

3.3 startdoc keyword

Keyword startdoc tells to Vesti that the main document starts. In the main document, you can also write startdoc in the main document. In that case, startdoc does nothing.

3.4 useenv keyword

As the name implies, keyword useenv is an analogous of \begin{...} and \end{...} pair in LaTeX. The simplest useenv is like this.

```
useenv center {
    Hello, World!
}
```

or

```
useenv center { Hello, World! }
```

As you can see, useenv center is the part of \begin{center}, and the single } is the part of \end{center}. Since Vesti knows their pair, one can write a code with several environment, and each pair is properly matched. For instance, above example is written in Vesti like follows. Here, \useenv just prints useenv in that style.

```
useenv figure [ht] {
    \centering
    useenv tikzpicture {
        useenv scope {
            \path (0,0) node {\vbox{
            %#\hbox{\tt\useenv center \{}
            %#\hbox{\tt\obeyspaces    Hello, World!}
            %#\hbox{\tt\obeyspaces\}}
            }};
        }
        \path (2.3,0) node {or};
        useenv scope [shift={(6,0)}] {
            \path (0,0) node {\tt\useenv center \{ Hello, World! \}};
        }
    }
}
```

Full syntax about useenv is the following.

$$\texttt{useenv}\ \sqcup\ \langle\text{environment name}\rangle\ \sqcup\ \langle\text{argument}\rangle^*\ \sqcup\ \texttt{\{}\ \langle\text{body}\rangle\ \texttt{\}}$$

where '*' means that the number of <argument> is zero or at least one, and

$$\langle\text{argument}\rangle = \begin{cases} (\langle\text{argument}\rangle) & \text{mandatory arguments} \\ [\langle\text{argument}\rangle] & \text{optional arguments} \end{cases}$$

For instance, below one is a valid Vesti code (environment `foo` is undefined in general). As one can see, spaces cannot exist in between <argument>s.

```
useenv foo (asd)(fff)[\ames and \awdsa](askws)[\rrsaa] {
    foobar
}
```

3.5 begenv and endenv keywords

As the name implies, both keywords begenv and endenv are analogous of `\begin{...}` and `\end{...}` pair in LaTeX, respectively. Thus below code

```
begenv center
    asdsad
endenv
```

is exactly same as

```
useenv center {
    asdsad
}
```

Then why we need begenv and endenv if we already have useenv? The only reason which both begenv and endenv! exist is defining new environment. For instance, one defines a new environment like following.

```
defenv foo {begenv minipage(\textwidth)}{endenv}
```

3.6 defun keyword

defun is the keyword which makes a LaTeX function. Internally, it uses `xparse` package which is automatically included from modern LaTeX2ϵ kernel. The grammar of defun is the following.

$$\texttt{defun}\ \sqcup\ [\langle\text{attr}\rangle]?\ \sqcup\ \langle\text{command-name}\rangle\ \sqcup\ (\langle\text{param-spec}\rangle)?\ \sqcup\ \{\langle\text{implementation}\rangle\}$$

Here, `?` means that this is an optional.

First, let us see some examples for defun, and then explain the meaning of each fields.

```
\defun foo {Hello, World!}
```

4 Builtins

Vesti also has its own builtin functions, which are prefixed with #. One might wonder what distinguishes builtins from keywords. In fact, from the compiler's internal perspective, there is no real difference. However, in actual language usage, constantly typing the prefix can be somewhat tedious, especially for functions that are commonly used.

From the perspective of language design sometimes desirable to use names that can not be reserved as keywords. For example, Vesti provides a built-in function #label, which will be explained later. Since Vesti is a typewriting-oriented language, the word "label" is often used in its ordinary sense rather than in its special semantic meaning within the language.

Followings are reserved as builtin functions.

#at_off	#at_on	#chardef	#def	#enum
#enum_counter	#eq	#get_filepath	#label	#ltx3_off
#ltx3_on	#mathchardef	#mathmode	#noltx3	#picture
#raw_tex	#showfont	#textmode	#undef	

Table 2: Builtins in Vesti

Since fancyvrb does not allow verbatim-like commands inside of Verbatim environment, builtins are not colored in example codes.

4.1 #label builtin

For the compiling issue, if one label some environment, one should write like follows using LaTeX function \label.

```
useenv theorem { \label{eq:1}
    1 + 1 = 2.
}
```

However, such code is not natural. Thus #label comes into the place. One can label environments using #label builtin like this.

```
#label(eq:1) useenv theorem {
    1 + 1 = 2.
}
```

One can also write like the following, which I personally prefer.

```
#label(eq:1)
useenv theorem {
    1 + 1 = 2.
}
```

4.2 #eq builtin

#eq is a abbriviation of 'useenv equation'. Actually, below codes are same.

```
#label(eq:1)
useenv equation {
    \sum_{n=1}^oo {1//n^2} = {\pi^2//6}.
}
```

```
#eq(eq:1) {
    \sum_{n=1}^oo {1//n^2} = {\pi^2//6}.
}
```

Since Vesti is used in the mathematical documents mainly, small syntactic suger is needed, so Vesti introduce `#eq` builtin.

4.3 `#chardef` builtin

`#chardef` defines a text token using the unicode codepoint. Here is the grammar of `#chardef` builtin.

`#chardef` ␣ <unicode-codepoint> ␣ <function>

Here, <unicode-codepoint> must be hexadecimal.

4.4 `#mathchardef` builtin

`#mathchardef` defines a math token using the unicode codepoint. Here is the grammar of `#mathchardef` builtin.

`#mathchardef` ␣ <kind> ␣ <font-num> ␣ <unicode-codepoint> ␣ <function>

Here, <unicode-codepoint> must be hexadecimal. Before explaining each parameter, introduce some examples.

- `#mathchardef` ␣ .opening ␣ 0 ␣ 29d8 ␣ `\lfooo`
- `#mathchardef` ␣ .ordinary ␣ 0 ␣ 2202 ␣ `\diff`

4.5 `#enum` and `#enum_counter` builtins

`#enum` builtin is just an enumerate environment with minimal support of `enumitem` feature. For example,

```
#enum {
    \item aaa
    \item bbb
    \item ccc
}
```

prints

1. aaa
2. bbb
3. ccc

`#enum` also can be nested like follows.

```
#enum {
    \item aaa
    \item #enum {
        \item bbb
        \item ccc
    }
    \item ddd
}
```

It prints

1. aaa
2. (a) bbb

(b) ccc

3. ddd

To customize a format of `\item` like `enumitem`, one can write like the following:

```
#enum (\Roman*.) {
    \item aaa
    \item #enum ({\roman*}**) {
        \item bbb
        \item ccc
    }
    \item ddd
}
```

I. aaa

II. i* bbb

ii* ccc

III. ddd

As one can see, `#enum` changes the format with inside of `()`. In Vesti, single `*` changes into `{<enum count>}` (including braces), and consequtive `**` changes into single `*` character. Thus, `\Roman*.` changes into `\Roman{enumi}.` and `{\roman*}**` changes into `{\roman{enumii}}*` (because this part is inside of another `#enum`).

Beware that by the parting mechanism, `\roman***` changes into `\roman*{<enum count>}`, and `\roman{*}**` into `\roman{{<enum count>}}*`, which are invalid arguments for `\roman` LaTeX internal command.

`#enum_counter` builtin is a helper to get the name of the current enum counter. For example, if one wants to start a second enum from 3, one can write

```
#enum {
    \item aaa
    \item #enum {
        \setcounter{#enum_counter}{2}
        \item bbb
        \item ccc
    }
    \item ddd
}
```

and it prints

1. aaa
2. (c) bbb

   (d) ccc
3. ddd

4.6 `#get_filepath` builtin

Because of the Vesti internal, if one includes picture by using `\includegraphics` for instance, providing the raw path cause an error from LaTeX with "file not found". `#get_filepath` builtin adjust raw file path into the new relative one which LaTeX knows. Here is the example code.

```
\includegraphics{#get_filepath(./foo.png)}
```

Internally, `#get_filepath` changes file path into the relative one with respect to `.vesti-dummy`. For example, if `foo.png`, `foo.ves` and `.vesti-dummy` are located in the root directory, then inside of `foo.ves`, `#get_filepath(./foo.png)` will changed into `../foo.png` because for `.vesti-dummy`, `foo.png` is located outside of it.

4.7 #at_on and #at_off builtins

Both builtins are same as LaTeX functions `\makeatletter` and `\makeatother` plus changes Vesti lexer such that `@` character is also can be a LaTeX function name. For instance, without using `#at_on` command, below code does not compiles.

```
defun @foo (#1) {Hello, #1!}
```

This is because `defun` expects a valid LaTeX function name but the character `@` is not a valid one except after the `\makeatletter` function. Since `defun` is a Vesti grammar, there is no way to tell Vesti that `@` is a right one although one uses `\makeatletter`. By using `#at_on`, below code then compiles.

```
#at_on
defun @foo (#1) {Hello, #1!}
#at_off
```

One can not using `#at_off`, but I strongly recommand to match pairs.

4.8 #ltx3_on and #ltx3_off builtins

Those builtins are similar with `#at_on` and `#at_off` pairs but for LaTeX3.

4.9 #textmode and #mathmode builtins

Internally, vesti tracks so called `text mode` and `math mode`. In the math mode, for instance, the token `->` is changed into → inside of the math mode. The idea is that Vesti changes `->` into `\to`, and so on. But when defining some function using `defun`, for example, one might want to define

```
useenv foo {-><-}
```

so that `$\foo$` makes →←, one should change the mode. Here, `#mathmode` comes into the place. One should write instead that

```
useenv foo {#mathmode{-><-}}
```

5 Lua API

Vesti uses Lua in both building script and inline inside of Vesti code. Below are functions which are baked in Vesti compiler.

compile	download	getCurrentDir	getEngineType
getModule	joinpath	mkdir	parse
ping	print	setCurrentDir	unzip
vestiDummyDir			

Table 3: Lua builtin functions in Vesti

6 Source Code of This Document

Below code was generated by inline lua.

```
docclass article (10pt)
importpkg {
    geometry (a4paper, margin = 2.2cm),
    xcolor,
    tikz,
    fancyvrb,
    amsmath,
}

\title{Vesti Transpiler User Manual}
\author{Sungbae Jeong}
#::
local function read_all(path)
  local f, err = io.open(path, "rb")
  assert(f, ("cannot open %s: %s"):format(path, err))
  local data = f:read("*a")
  f:close()
  return data
end
::#<readAll>

-- definition of \keyword command
defun [!] keyword (m) {{\tt\color{purple}#1}}
defun [!] builtin (v) {{\tt\color{blue!65!yellow}\##1}}
#::
local commands = { "useenv", "defun", "defenv" }
for _, command in ipairs(commands) do
    local s = string.format(
        "defun %s (s) {\\IfBooleanTF{#!1}{\\keyword{%%!-%s-%%}}{\\keyword{%%!-%s-%%} }}",
        command, command, command
    )
    vesti.print(vesti.parse(s))
end
::#

#::
function begenv(add_bang, add_space)
    local name = "begenv"
    if add_bang then name = name .. "!" end
    local begenv = "\\keyword{" .. name .. "}"
    if add_space then begenv = begenv .. " " end
    vesti.print(begenv, { nl = 0 })
end

function endenv(add_bang, add_space)
    local name = "endenv"
    if add_bang then name = name .. "!" end
    local endenv = "\\keyword{" .. name .. "}"
    if add_space then endenv = endenv .. " " end
    vesti.print(endenv, { nl = 0 })
end
::#*

-- space character used in the texbook.
#at_on
#chardef 2423 \@b
```

```
defun ob {\kern2pt{\tt\@b}\kern2pt}
#at_off

startdoc
\maketitle
\tableofcontents

\section{Introduction}
I used to create several documents using \LaTeX\ (or plain \TeX\ but \TeX\ is
quite cumbersome to write-especially when working with very complex tables or
inserting images). Its markdown-like syntax is also not comfortable to use. For
example, here is a simple \LaTeX\ document:

useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
% coprime is my custom class. See https://github.com/e0328eric/coprime.
\documentclass[tikz, geometry]{coprime}

\settitle{My First Document}{Sungbae Jeong}{}
\setgeometry{a4paper, margin = 2.5cm}

\begin{document}
\section{Foo}
Hello, World!
\begin{figure}[ht]
    \centering
    \begin{tikzpicture}
        \draw (0,0) -- (1,1);
    \end{tikzpicture}
\end{figure}

The code above is a figure using TikZ.

\end{document}
-%}

What annoys me most when using it is the \lq\verb@\begin@\rq\ and
\lq\verb@\end@\rq\ blocks. Is there a way to write something much simpler? This
question led me to start this project. Currently, the following code is
generated into the \LaTeX code above (except comments) using vesti:

useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
% coprime is my custom class. See https://github.com/e0328eric/coprime.
+keyword|docclass@ coprime (tikz, geometry)

\settitle{My First Document}{Sungbae Jeong}{}
\setgeometry{a4paper, margin = 2.5cm}

+keyword|startdoc@

\section{Foo}
Hello, World!
+keyword|useenv@ figure [ht] {
    \centering
    +keyword|useenv@ tikzpicture {
        \draw (0,0) -- (1,1);
    }
}

The code above is a figure using TikZ.
-%}
```

```
117
118 Everywhere in this paper, the symbol \ob\ denotes a \lq\lq space\rq\rq\ when
119 one use Vesti.
120
121 \section{Structure of Vesti File}
122 Vesti is similar as \LaTeX. Its structure consists with two parts: {\tt preamble} and
123 {\tt main}. Preamble is the place where \LaTeX\ documentclass, packages, and
124 several settings are located. Main body is where actual documentation is located.
125 Below figure is the simple Vesti documentation.
126
127 useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
128 +keyword|docclass@ article (10pt)
129 +keyword|importpkg@ {
130     geometry (a4paper, margin=2.2cm)
131 }
132 +keyword|startdoc@
133 Hello, Vesti!
134 -%}
135
136 We will see later, but the very difference with \LaTeX\ is that Vesti has its
137 own keywords (keywords are colored with purple). It makes the code readable and
138 it is easier and faster to write the document. The keyword startdoc splits
139 the preamble and the main part of the documentation similar with
140
141 -- Don't ask why I chose Q for catcode 0.
142 %#{\tt\catcode`Q=0 Qcatcode`\\=12 \beginQ{documentQ}} in \LaTeX.
143 However, Vesti does not have the analogous part of
144 %#{\tt\catcode`Q=0 Qcatcode`\\=12 \endQ{documentQ}},
145 because almost every \LaTeX\ document (99.999\% I'm sure) does not have any code
146 below %#{\tt\catcode`Q=0 Qcatcode`\\=12 \endQ{documentQ}}.
147 For this reason, Vesti automatically ends document when EOF (End Of File) is
148 found.
149
150 \section{Keywords}
151 Followings are reserved as keywords. In this document, every Vesti keyword has
152 the form like \keyword{this}.
153 useenv table [ht] {
154     \centering
155     #::
156     local content = read_all("../src/lexer/Token.zig")
157
158     -- Lua's built-in patterns don't support lookahead.
159     -- We capture both the keyword and the TokenType, then filter out 'deprecated'.
160     -- Pattern breakdown:
161     --   %.%{         => matches ".{"
162     --   %s*"([^"]+)" => a quoted string -> capture 1
163     --   %s*,%s*TokenType%.([%w_]+) => TokenType.<Name> -> capture 2
164     local pat = "%.%{%s*\"([^\"]+)\"%s*,%s*TokenType%.([%w_]+)"
165
166     local keywords = {}
167     for name, tok in content:gmatch(pat) do
168       if tok ~= "deprecated" then
169         keywords[#!keywords + 1] = name
170       end
171     end
172
173     table.sort(keywords)
174
175     vesti.print([[\begin{tabular}{cccc}]])
176
```

```
    for i, kw in ipairs(keywords) do
        local cell = string.format("\\keyword{%s}", kw)
        if (i % 4) == 0 then
            vesti.print(cell .. [[\\]])
        else
            vesti.print(cell .. "&")
        end
    end

    vesti.print([[\end{tabular}]])
    ::#[readAll]
    \caption{Keywords in Vesti}
}

\subsection{\keyword{docclass} keyword}
Keyword \keyword{docclass} is an analogous of \verb|\documentclass| in \LaTeX.
If \keyword{docclass} is in the main paragraph, it acts just a normal word.
In other words, \keyword{docclass} actives only in the preamble.
The syntax of \keyword{docclass} is following:

useenv center {
    \keyword{docclass}\ob<class name>\ob{\tt(}<arguments>{\tt)}
}
Here, arguments are separated by commas and embraced by {\tt ()}. Here are some
examples.

\goodbreak
useenv itemize {
    \item \keyword{docclass}\ob{\tt article}
    \item \keyword{docclass}\ob{\tt article\ob(10pt)}
    \item \keyword{docclass}\ob{\tt article\ob(10pt,\ob twocols)}
    \item \keyword{docclass}\ob{\tt article\ob(10pt,twocols)}
}

\subsection{\keyword{importpkg} keywords}
Keyword \keyword{importpkg} is an analogous of \verb|\usepackage| in \LaTeX.
If \keyword{importpkg} is in the main paragraph, it acts just a normal word.
In other words, \keyword{importpkg} actives only in the preamble.

importpkg has two different syntax. First one is same as docclass.
useenv center {
    \keyword{importpkg}\ob<pkg-name>\ob{\tt(}arguments{\tt)}
}
Here, arguments are separated by commas and embraced by {\tt ()}.
In the practical case, one should include several packages with options.
importpkg also supports such case. We will look at an example instead of
giving rigorous grammar.
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
+keyword|importpkg@ {
    amsmath, amssymb, amsthm,
    geometry (a4paper, margin=2.2cm),
}
-%}

\noindent As one can see, inside of \verb|{}|, several packages can be used
together with thier options.

\subsection{\keyword{startdoc} keyword}
Keyword \keyword{startdoc} tells to Vesti that the main document starts. In the
main document, you can also write \keyword{startdoc} in the main document. In
```

```
that case, \keyword{startdoc} does nothing.

\subsection{\useenv keyword}
As the name implies, keyword \useenv is an analogous of \verb|\begin{...}| and
\verb|\end{...}| pair in \LaTeX.
The simplest \useenv is like this.

useenv figure [ht] {
    \centering
    useenv tikzpicture {
        useenv scope {
            \path (0,0) node {\vbox{
            %#\hbox{\tt\useenv center \{}
            %#\hbox{\tt\obeyspaces    Hello, World!}
            %#\hbox{\tt\obeyspaces\}}
            }};
        }
        \path (2.3,0) node {or};
        useenv scope [shift={(6,0)}]{
            \path (0,0) node {\tt\useenv center \{ Hello, World! \}};
        }
    }
}

As you can see, {\tt\useenv center} is the part of \verb|\begin{center}|, and
the single {\tt\}} is the part of \verb|\end{center}|. Since Vesti knows their
pair, one can write a code with several environment, and each pair is properly
matched. For instance, above example is written in Vesti like follows. Here,
\verb|\useenv| just prints \useenv in that style.

useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
|+color|purple@useenv@ figure [ht] {
    \centering
    |+color|purple@useenv@ tikzpicture {
        |+color|purple@useenv@ scope {
            \path (0,0) node {\vbox{
            |+color|blue@%#@\hbox{\tt\useenv center \{}
            |+color|blue@%#@\hbox{\tt\obeyspaces    Hello, World!}
            |+color|blue@%#@\hbox{\tt\obeyspaces\}}
            }};
        }
        \path (2.3,0) node {or};
        |+color|purple@useenv@ scope [shift={(6,0)}] {
            \path (0,0) node {\tt\useenv center \{ Hello, World! \}};
        }
    }
}
-%}

Full syntax about \useenv is the following.
useenv center {
    \useenv\ob<environment name>\ob<argument>*\ob{\tt\{} <body> {\tt\}}
}
where `*' means that the number of <argument> is zero or at least one, and
$$
    "<argument>" = useenv cases {
        "(<argument>)" & "mandatory arguments" \cr
        "[<argument>]" & "optional arguments"  \cr
    }
$$
```

```
For instance, below one is a valid Vesti code (environment {\tt foo} is
undefined in general). As one can see, spaces cannot exist in between <argument>s.
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
|+color|purple@useenv@ foo (asd)(fff)[\ames and \awdsa](askws)[\rrsaa] {
    foobar
}
-%}

#def #begenv {#::begenv(false,true)::#*}
#def #begenv_arg {#::begenv(#1,#2)::#*}
#def #endenv {#::endenv(false,true)::#*}
#def #endenv_arg {#::endenv(#1,#2)::#*}

\subsection{#begenv and #endenv keywords}
As the name implies, both keywords #begenv and #endenv are
analogous of \verb|\begin{...}| and \verb|\end{...}| pair in \LaTeX,
respectively. Thus below code
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
|+color|purple@begenv@ center
    asdsad
|+color|purple@endenv@
-%}
is exactly same as
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
+keyword|useenv@ center {
    asdsad
}
-%}
Then why we need #begenv and #endenv if we already have \useenv*?
The only reason which both #begenv and #endenv_arg(true)(false) exist is
defining new environment. For instance, one defines a new environment like
following.
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@] {
%#+keyword|defenv@ foo {|+color|purple@begenv@ minipage(\textwidth)}{|+color|purple@endenv@}
}

\subsection{\defun keyword}
\defun is the keyword which makes a \LaTeX\ function. Internally, it uses {\tt
xparse} package which is automatically included from modern \LaTeX$2\epsilon$
kernel. The grammar of \defun is the following.

begenv center
    \defun\ob[<attr>]?\ob<command-name>\ob(<param-spec>)?\ob\{<implementation>\}
endenv
Here, {\tt`?`} means that this is an optional.

First, let us see some examples for \defun*, and then explain the meaning of
each fields.

#::
local examples = {
    "\\defun foo {Hello, World!}"
}

for _, example in ipairs(examples) do
    vesti.print(vesti.parse(
    "begenv! Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]"
    ))
    vesti.print(example)
    vesti.print(vesti.parse("endenv! Verbatim"))
```

```
end
::#

\section{Builtins}
Vesti also has its own builtin functions, which are prefixed with \#.
One might wonder what distinguishes builtins from keywords. In fact, from the
compiler's internal perspective, there is no real difference. However, in actual
language usage, constantly typing the prefix can be somewhat tedious, especially
for functions that are commonly used.

From the perspective of language design --particularly in Vesti-- it is
sometimes desirable to use names that can not be reserved as keywords. For example,
Vesti provides a built-in function \builtin|label|, which will be explained
later. Since Vesti is a typewriting-oriented language, the word \lq\lq
label\rq\rq\ is often used in its ordinary sense rather than in its special
semantic meaning within the language.

Followings are reserved as builtin functions.

useenv table [ht] {
    \centering
    #::
    local content = read_all("../src/lexer/Token.zig")

    -- match .{ "here" }
    local pat = "%.%{%s*\"([^\"]+)\"%s*%}"

    local builtins = {}
    for name, tok in content:gmatch(pat) do
        builtins[#!builtins + 1] = name
    end
    table.sort(builtins)

    vesti.print([[\begin{tabular}{ccccc}]])

    for i, kw in ipairs(builtins) do
        local cell = string.format("\\builtin@%s@", kw)
        if (i % 5) == 0 then
            vesti.print(cell .. [[\\]])
        else
            vesti.print(cell .. "&")
        end
    end

    vesti.print([[\end{tabular}]])
    ::#[readAll]
    \caption{Builtins in Vesti}
}
Since {\tt fancyvrb} does not allow verbatim-like commands inside of
{\tt Verbatim} environment, builtins are not colored in example codes.

\subsection{\builtin|label| builtin}
For the compiling issue, if one label some environment, one should write like
follows using \LaTeX\ function \verb|\label|.
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=&|@]{%-
&keyword|useenv@ theorem { \label{eq:1}
    1 + 1 = 2.
}
-%}

```

```
However, such code is not natural. Thus \builtin|label| comes into the place.
One can label environments using \builtin|label| builtin like this.
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=&|@]{%-
#label(eq:1) &keyword|useenv@ theorem {
    1 + 1 = 2.
}
-%}
One can also write like the following, which I personally prefer.
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=&|@]{%-
#label(eq:1)
&keyword|useenv@ theorem {
    1 + 1 = 2.
}
-%}

\subsection{\builtin|eq| builtin}
\builtin|eq| is a abbriviation of \lq\keyword{%-useenv-%} equation\rq.
Actually, below codes are same.

useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
#label(eq:1)
+keyword|useenv@ equation {
    \sum_{n=1}^oo {1//n^2} = {\pi^2//6}.
}
-%}
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
#eq(eq:1) {
    \sum_{n=1}^oo {1//n^2} = {\pi^2//6}.
}
-%}
Since Vesti is used in the mathematical documents mainly, small syntactic suger
is needed, so Vesti introduce \builtin|eq| builtin.

\subsection{\builtin|chardef| builtin}
\builtin|chardef| defines a text token using the unicode codepoint.
Here is the grammar of \builtin|chardef| builtin.
useenv center {
    \builtin|chardef|\ob<unicode-codepoint>\ob<function>
}
Here, <unicode-codepoint> must be hexadecimal.

\subsection{\builtin|mathchardef| builtin}
\builtin|mathchardef| defines a math token using the unicode codepoint.
Here is the grammar of \builtin|mathchardef| builtin.
useenv center {
    \builtin|mathchardef|\ob<kind>\ob<font-num>\ob<unicode-codepoint>\ob<function>
}
Here, <unicode-codepoint> must be hexadecimal.
Before explaining each parameter, introduce some examples.
useenv itemize {
    \item \builtin|mathchardef|\ob.opening\ob0\ob29d8\ob\verb|\lfooo|
    \item \builtin|mathchardef|\ob.ordinary\ob0\ob2202\ob\verb|\diff|
}

\subsection{\builtin|enum| and \builtin|enum_counter| builtins}
\builtin|enum| builtin is just an enumerate environment with minimal support of
{\tt enumitem} feature. For example,
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
#enum {
    \item aaa
```

```
    \item bbb
    \item ccc
}
-%}
prints
#enum {
    \item aaa
    \item bbb
    \item ccc
}
\builtin|enum| also can be nested like follows.
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
#enum {
    \item aaa
    \item #enum {
        \item bbb
        \item ccc
    }
    \item ddd
}
-%}
It prints
#enum {
    \item aaa
    \item #enum {
        \item bbb
        \item ccc
    }
    \item ddd
}
To customize a format of \verb|\item| like {\tt enumitem}, one can
write like the following:
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
#enum (\Roman*.) {
    \item aaa
    \item #enum ({\roman*}**) {
        \item bbb
        \item ccc
    }
    \item ddd
}
-%}
#enum (\Roman*.) {
    \item aaa
    \item #enum ({\roman*}**) {
        \item bbb
        \item ccc
    }
    \item ddd
}
As one can see, \builtin|enum| changes the format with inside of {\tt()}.
In Vesti, single {\tt *} changes into {\tt \{<enum count>\}} (including braces),
and consequtive {\tt **} changes into single {\tt *} character.
Thus, \verb|\Roman*.| changes into \verb|\Roman{enumi}.| and \verb|{\roman*}**|
changes into \verb|{\roman{enumii}}*| (because this part is inside of another
\builtin|enum|).

Beware that by the parting mechanism, \verb|\roman***| changes into
\verb|\roman*{<enum count>}|, and \verb|\roman{*}**| into \verb|\roman{{<enum count>}}*|,
which are invalid arguments for \verb|\roman| \LaTeX\ internal command.
```

```
\builtin|enum_counter| builtin is a helper to get the name of the current enum
counter. For example, if one wants to start a second enum from 3, one can write
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
#enum {
    \item aaa
    \item #enum {
        \setcounter{#enum_counter}{2}
        \item bbb
        \item ccc
    }
    \item ddd
}
-%}
and it prints
#enum {
    \item aaa
    \item #enum {
        \setcounter{#enum_counter}{2}
        \item bbb
        \item ccc
    }
    \item ddd
}

\subsection{\builtin|get_filepath| builtin}
Because of the Vesti internal, if one includes picture by using
\verb|\includegraphics| for instance, providing the raw path cause an error from
\LaTeX\ with \lq\lq file not found\rq\rq. \builtin|get_filepath| builtin adjust
raw file path into the new relative one which \LaTeX\ knows. Here is the example
code.
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@] {
%#\includegraphics{#get_filepath(./foo.png)}
}
Internally, \builtin|get_filepath| changes file path into the relative one with
respect to {\tt.vesti-dummy}. For example, if {\tt foo.png}, {\tt foo.ves}
and {\tt.vesti-dummy} are located in the root directory, then inside of {\tt
foo.ves}, \builtin|get_filepath|{\tt(./foo.png)} will changed into
{\tt../foo.png} because for {\tt.vesti-dummy}, {\tt foo.png} is located outside
of it.

\subsection{\builtin|at_on| and \builtin|at_off| builtins}
Both builtins are same as \LaTeX\ functions \verb|\makeatletter| and
\verb|\makeatother| plus changes Vesti lexer such that \verb|@| character is
also can be a \LaTeX\ function name. For instance, without using \builtin|at_on|
command, below code does not compiles.
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|&] {
%#+keyword|defun& @foo (#1) {Hello, #1!}
}
This is because \keyword{%-defun-%} expects a valid \LaTeX\ function name but
the character \verb|@| is not a valid one except after the \verb|\makeatletter|
function. Since \keyword{%-defun-%} is a Vesti grammar, there is no way to tell
Vesti that \verb|@| is a right one although one uses \verb|\makeatletter|.
By using \builtin|at_on|, below code then compiles.
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|&]{%-
#at_on
+keyword|defun& @foo (#1) {Hello, #1!}
#at_off
-%}
One can not using \builtin|at_off|, but I strongly recommand to match
```

```
pairs.

\subsection{\builtin|ltx3_on| and \builtin|ltx3_off| builtins}
Those builtins are similar with \builtin|at_on| and \builtin|at_off|
pairs but for \LaTeX3.

\subsection{\builtin|textmode| and \builtin|mathmode| builtins}
Internally, vesti tracks so called {\tt text mode} and {\tt math mode}.
In the math mode, for instance, the token \verb|->| is changed into $->$ inside
of the math mode. The idea is that Vesti changes \verb|->| into \verb|\to|, and
so on. But when defining some function using \defun*, for example, one might
want to define
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
+keyword|useenv@ foo {-><-}
-%}
so that \verb|$\foo$| makes $-><-$, one should change the mode. Here,
\builtin|mathmode| comes into the place. One should write instead that
useenv Verbatim [numbers=left, numbersep=5pt, frame=single, commandchars=+|@]{%-
+keyword|useenv@ foo {#mathmode{-><-}}
-%}

\section{Lua API}
Vesti uses Lua in both building script and inline inside of Vesti code.
Below are functions which are baked in Vesti compiler.
useenv table [ht] {
    \centering
    #::
    local content = read_all("../src/Lua.zig")

    local pat = '%.name%s*=%s*"([^"]+)"'

    local lua = {}
    for name, tok in content:gmatch(pat) do
        lua[#!lua + 1] = name
    end
    table.sort(lua)

    vesti.print([[\begin{tabular}{ccccc}]])

    for i, kw in ipairs(lua) do
        local cell = string.format("\\tt{%s}", kw)
        if (i % 4) == 0 then
            vesti.print(cell .. [[\\]])
        else
            vesti.print(cell .. "&")
        end
    end

    vesti.print([[\end{tabular}]])
    ::#[readAll]
    \caption{Lua builtin functions in Vesti}
}


\newpage
\section{Source Code of This Document}
Below code was generated by inline lua.
useenv Verbatim [numbers=left, numbersep=5pt, frame=single] {
#::
    local content = read_all("vesti_man.ves")
```

```
    for line in content:gmatch("([^\r\n]*)\r?\n?") do
        vesti.print(line)
    end
::#[readAll]
}

```