1 The Lepton manual

1.1 Tutorial

To write a "hello world" manuscript, we use the `hello.nw` file as an example of the LaTeX-like syntax:

Code chunk 1: «hello.nw»

```
\documentclass[paper=a7]{scrartcl}
\usepackage[width=7cm,height=10cm]{geometry}
\usepackage{float} \newfloat{leptonfloat}{H}{lol}
\begin{document}
The code below sends "hello world" instructions to the \verb ocaml  interpreter.
<<hello_world -exec ocaml -chunk ocaml>>=
let msg = "Hello world.";;
print_string(msg); print_newline();;
@
\end{document}
```

The Lepton executable splits the file into documentation and source code, executes instructions where specified, and embeds the results. Lepton turns `hello.nw` into a legitimate LaTeX document `hello.tex`. When processing a file, Lepton outputs the name of each encountered code snippet and how it deals with it.

Code chunk 2: «hello.tex»

```
./lepton.bin -format_with tex hello.nw -o hello.tex
```

Interpret with `shell`

```
This is the Lepton/Lex implementation.
hello_world (part 1): chunk as ocaml, exec with ocaml, output as text,
```

The `hello.tex` file is compiled with `pdflatex`. The resulting PDF file is displayed on the right.

Code chunk 3: «hello.pdf»

```
pdflatex -interaction=batchmode hello.tex
```

Interpret with `shell`

```
This is pdfTeX, Version 3.14159265-2.6-1.40.21 (TeX Live 2020/Debian) (preloaded format=pdflatex)
 restricted \write18 enabled.
entering extended mode
```

The code below sends "hello world" instructions to the `ocaml` interpreter.

```
let msg = "Hello world.";;
print_string(msg); print_newline();;
```

Interpret with `ocaml`

```
val msg : string = "Hello world."
Hello world.
- : unit = ()
```

leptonfloat 1: hello_world

1.2 Usage and command-line options

```
lepton [-format_with formatter] [filename] [-o output]
```

By default, Lepton reads from `stdin`, writes to `stdout` and formats chunks in LaTeX format with the `minted` package for pretty-printing (see 1.5 for details). Provided options are set in appearing order, with the following effects :

- **filename** sets the input file name.
- **-o output** sets the name of the generated documentation file.
- **-format_with formatter** sets the `formatter` for embedding chunk contents and the output of executable instructions in the documentation file.

1.3 Syntax

In the spirit of literate programming [2], Lepton files are written in a documentation format such as LaTeX, HTML or Wiki markup with special blocks called *code chunks*.

Similar to Noweb files [4], code chunks start with a chunk header of the form `<<header>>=` at the beginning of the line, and end with `@` at the beginning of the line. Lepton parses the chunk header as a blank separated command line, and the first word is treated as the chunk name. The following words are interpreted as chunk options. These control the output and interpretation of the chunk contents. See Section 1.4 for further details.

Code chunks contain any type of textual bits and pieces, including source code, input data, executable instructions and nested code chunks. This allows embedding Lepton files inside other Lepton files, such as the `hello.nw` example. Inside a code chunk, `@@` at the beginning of a line is replaced by a single `@`, but not for nested chunks.

Lepton does not alter the contents of the input file, except for the following directives :

- The chunk header is formatted into the selected documentation format.
- A series of blanks followed by `<<chunkname>>` inside a code chunk represents a chunk reference, and is expanded to the contents of the code chunk `chunkname`.
- `\Linput{filename}` at the beginning of a line outside a code chunk is replaced by the contents of the file. This is performed before interpretation, so everything defined in `filename` is available ; code chunks can be executed and can be referenced.
- `\Lexpr{interpreter}{code}` outside a code chunk is used to directly embed the results of sending the `code` as commands to the `interpreter`. This can be used to include the value of variables or results in the text.

Code chunks can be divided into small meaningful entities that are easy to document. Code chunks can be written in several parts. Options defined in the chunk header are propagated to the following parts.

Chunk references are replaced by the concatenation of all chunks with the same name, including the recursive references. The amount of whitespace before the chunk reference is used to set the indentation level: it is prepended to all lines when expanding the reference.

N.B. Characters appearing on the same line after a chunk header, a chunk end, a chunk reference, a `\Linput` are ignored and can be used for comments.

1.4 Interpretation of code snippets

The contents of code chunks are interpreted as specified by the options in the chunk header:

- `-write -nowrite` : write the chunk contents to disk and use the chunk name as file name. Default: `-nowrite`,
- `-expand -noexpand` : expand chunk references in the documentation. Default: `-noexpand`,
- `-exec interpreter` : execute the chunk contents in an external interpreter. Default: `none`, i.e. do not execute,
- `-chunk format -output format` : indicate the format of chunk contents and chunk output for pretty-printing (see Section 1.5).

Lepton interprets the source file sequentially. For each chunk, the references are recursively expanded, then the chunk contents are optionally written to disk, and the chunk contents are optionally sent to the external interpreter. In particular, written files and definitions sent to an interpreter are available for the subsequent code chunks. When launched in a terminal, Lepton displays the chunk names, and the options used to process them.

When writing to disk, relative paths and full paths can be used for the file name. However, Lepton does not create the parent directories when absent.

The `interpreter` specified with `-exec` or `\Linput` is a session / process name. If it corresponds to a process already open by Lepton, the process will be reused. Otherwise, the interpreter name is matched (by prefix) to a list of known intepreters and a new instance is launched. Lepton currently supports the UNIX shell, OCaml, Python, and R. Several sessions of the same process can be open concurrently, e.g. `shell1, shell2, shellbis`. Note that Lepton catches the input and output of interpreters, so programs cannot be used interactively (programs launched by Lepton cannot wait for user input).

Other programming languages, notably compiled languages such as C/C++, can be used in Lepton by writing the source code to disk and using the `shell` interpreter to compile and execute the programs. To use a makefile, put the text into a chunk, write the chunk to disk and execute with `shell`.

Options that are set for a code chunk are propagated to the following chunks of the same name. `lepton_options` is a reserved chunk name for setting default options, the chunk contents are ignored. For example, `<<lepton_options -write -chunk ocaml>>=` sets the default behavior to writing all chunk contents to disk, and formatting the chunk contents as OCaml code.

1.5 Formatting

The `formatter` is responsible for presenting the contents of code chunks and their results in a format compatible with the documentation format. For instance, it packs source code in a verbatim environment for LaTeX or inside `<pre></pre>` tags for HTML. Chunk contents and chunk output are independently formatted according to their respective options.

A `formatter` is implemented as a function that receives the chunk name, options, the chunk contents and the output and produces some text to be included in the documentation file. Lepton includes the `latex_minted` formatter for inclusion in LaTeX and code pretty-printing with Pygments, the `tex` formatter for inclusion in LaTeX and inclusion of code in a `verbatim` environment, as well as the `html` and `creole` formatters for HTML and Wiki markup.

The predefined formatters recognize special values of the output format: `verb` (the output is already formatted and intended for direct inclusion) and `hide` (the output is not included). For pretty-printing in LaTeX, we use the `minted` package in combination with the Python `Pygments` beautifier [1] to provide colorful syntax highlighting for many languages. The `latex_minted` formatter wraps the chunk contents and its output in a `leptonfloat` environment, which is based on the `float` package (see below). Additionally,

- a caption is automatically included based on the chunk name,
- labels and indexes are automatically defined, the `hyperref` package can be used to link to chunk definitions,
- for each chunk reference, Lepton automatically adds a hyperlink to the corresponding chunk definition.

A list of all code chunks can be generated with `\lelistoflistings` and an index of code chunks with `makeidx`. These additions to LaTeX are defined in the `lepton.sty` file.

This is the LaTeX code produced by the `tex` formatter from the `hello_world` chunk in the tutorial.

```
\begin{leptonfloat}
\caption{hello\_world}
\label{hello_world}\vspace*{-\leptonlb}\footnotesize{\texttt{}}\vspace*{-\leptonlc}
\begin{verbatim}
let␣msg␣=␣"Hello␣world.";;
print_string(msg);␣print_newline();;
\end{verbatim}
\vspace*{-\leptonld}Interpret␣with␣\texttt{ocaml}\vspace*{-\leptonle}
\begin{verbatim}
val␣msg␣:␣string␣=␣"Hello␣world."
Hello␣world.
-␣:␣unit␣=␣()
\end{verbatim}
\end{leptonfloat}
```

1.6 Current implementation and availability

Lepton can be downloaded from Zenodo [3], and can be compiled with Ocaml. Note that some command interpreters may not work on all platforms, as they require functionality from Ocaml's Unix library which may be unavailable. The software repository is on Github, please report issues there.

1.7 Frequently Asked Questions

Gobble error is produced when Lepton when chunk nesting is incorrect. This usually happens when forgetting to close a chunk with `@`.

A warning is issued when a reference to a code chunk is not found anywhere in the document.

Python 3 migration : the name of the python executable and the usage of the print function are hardcoded in the current implementation. However Python 3 introduced deliberate incompatibilities. Currently users of the Python interpreter in code chunks must explicitly indicate the python version in the chunk header via `-exec python2` or `-exec python3`, and should not use syntax compatibility layers such as python-future. Currently the list of external interpreters is specified at compile time.

Dialog between Lepton and external interpreters happens by sending and catching print instructions. Reading from the process pipe when there is no output will typically result in `Fatal error: exception End_of_file`. This occurs when the interpreter is not install in the host system, or when the chunk content is malformed (syntax error). Some interpreters wait for a complete input, in which case the Lepton process seems stalled.

Beware of DOS end-of-line characters in UNIX files. Lepton will send them to the interpreter.

References

[1] Georg Brandl, Tim Hatch, and Armin Ronacher. *Pygments*. URL `http://pygments.org/`.

[2] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 01 1984. ISSN 0010-4620. doi: 10.1093/comjnl/27.2.97. URL `https://doi.org/10.1093/comjnl/27.2.97`.

[3] Sébastien Li-Thiao-Té. lepton: v1.0, July 2018. URL `https://doi.org/10.5281/zenodo.1311588`.

[4] Norman Ramsey. Literate programming simplified. *IEEE Softw.*, 11(5):97–105, September 1994. ISSN 0740-7459. doi: 10.1109/52.311070. URL `https://doi.org/10.1109/52.311070`.