Lepton implementation details

Li-Thiao-Té Sébastien

December 20, 2018

> One Ring to rule them all,
> One Ring to find them,
> One Ring to bring them all
> and in the darkness bind them

The Lord of the Rings, JRR Tolkien

1 Introduction

1.1 Overview

Lepton is an automaton for the literate execution of programs. As in the literate programming paradigm, Lepton makes it possible to reorganize source code in the form of meaningful chunks, regardless of the constraints of the programming language. In addition, Lepton files are executable programs which can:

- generate the complete hierarchy of source files in a software project,
- configure the environment, compile the source code and produce binary executables,
- execute the compiled programs and communicate with command interpreters to process data and produce figures,
- generate full-featured documentation for source code and executable instructions.

This manuscript contains the implementation details of Lepton, documented as a Lepton file, with many helpful comments on the programming techniques used in the source code. In addition, the program specifications included in this document are used to produce the standalone PDF manual and a tutorial. The software is published under the CeCILL-2.1 license.

Code chunk 1: «boilerplate»

```
(*
Copyright Li-Thiao-Té Sébastien (2018)
lithiao@math.univ-paris13.fr
This file was generated from lepton.nw by lepton.bin

This software is a computer program whose purpose is to facilitate the
creation and distribution of literate executable papers. It processes
files containing source code and documentation, and can execute commands
to produce (scientific) reports.

This software is governed by the CeCILL  license under French law and
abiding by the rules of distribution of free software.  You can  use,
modify and/ or redistribute the software under the terms of the CeCILL
license as circulated by CEA, CNRS and INRIA at the following URL
"http://www.cecill.info".

As a counterpart to the access to the source code and  rights to copy,
modify and redistribute granted by the license, users are provided only
with a limited warranty  and the software's author,  the holder of the
economic rights,  and the successive licensors  have only  limited
liability.

In this respect, the user's attention is drawn to the risks associated
with loading,  using,  modifying and/or developing or reproducing the
software by the user in light of its specific status of free software,
that may mean  that it is complicated to manipulate,  and  that  also
therefore means  that it is reserved for developers  and  experienced
professionals having in-depth computer knowledge. Users are therefore
encouraged to load and test the software's suitability as regards their
requirements in conditions enabling the security of their systems and/or
data to be ensured and,  more generally, to use and operate it in the
same conditions as regards security.

The fact that you are presently reading this means that you have had
knowledge of the CeCILL license and that you accept its terms.
*)
```

1.2 Document structure

This document is structured as a standard LaTeX file — with sections, subsections, etc. — with embedded chunks that may contain any type of textual data, including source code and executable instructions. Lepton processes the file in linear order; in particular, chunks that contain source code are written to disk, and instructions are executed where they are defined.

Think of this as a self-contained executable script intended to produce the Lepton executable program. The script follows these steps:
1. clean the current directory,

Code chunk 2: «clean»

```
# Remove pre-existing files, output is empty on success
rm *.ml *.bin *.pdf *.mll *.cmi *.cmo
rm lepton_manual.* hello.*
```

Interpret with `shell`

```
```

2. write the source code (Section 2)
3. compile the source code and run some tests (Section 3)
4. assemble the specifications into a standalone manual and compile it to PDF (Section 4)

2 Implementation

The implementation published in this document is written in the OCaml programming language [3] using only the standard library. It is divided into several modules:
- the `main` function,
- the `lepton` module contains type definitions and useful functions,
- the `lexer` module defines the syntax of Lepton files and the functions to process it with the `ocamllex` lexical analyzer,
- the `interpreters` module handles communication with external interpreters,
- and the `formatters` module prepares the contents of chunks and the output of command interpreters for inclusion in the documentation.

2.1 Main function

Specification

```
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 2.5 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.

The main function is responsible for
- parsing the command line options via the `Arg` library,

Code chunk 3: «main.ml»

boilerplate

```
<<boilerplate>>
print_endline "This is the Lepton/Lex implementation.";;
open Lepton;;      (* Load common definitions *)
open Interpreters;; (* Load the default set of interpreters *)
let option_spec = ("-o", Arg.String (fun s -> lepton_oc := open_out s), "name of the output file (default is stdout)") ::
                  ("-format_with", Arg.String Formatters.set, "set the formatter (default is LaTeX/minted") :: [] in
let anon_spec =   fun s -> lepton_ic := open_in s; in (* anonymous arguments are interpreted as input filename *)
Arg.parse option_spec anon_spec "usage: lepton [-format_with formatter] [filename] [-o output]";;
```

- calling the lexical analyzer and interpreting the chunk contents.

Code chunk 4: «main.ml (part 2)»

```
let chunks = Lexer.shabang (Lexing.from_channel !lepton_ic);;
let interpreter = function
  | Code (args, _) when args.(0) = "lepton_options" -> ignore(parse_chunklabel args); (* special chunk *)
  | Code (args, s) -> let option = parse_chunklabel args in
                      let plain, expanded = Lexer.expand "" "" (Lexing.from_string s) in
                      if option.write then send_to_file expanded args.(0);
                      let output = send_to_interpreter expanded option.interpreter in
                      if option.expand then !formatter args.(0) option [] expanded output
                      else !formatter args.(0) option (Lexer.chunkref_list (Lexing.from_string s)) plain output;
  | Doc s -> Lexer.lexpr (Lexing.from_string s);
in Queue.iter interpreter chunks;;
```

2.2 Lepton module

The `Lepton` module implements the common interfaces for all the other modules. It contains
- the type definitions,
- the type of options and related functions,
- the storage mechanism based on hash tables,

- communication channels with files and processes,
- variables for inter-module communication.

The lexical analyzer divides the Lepton file in a series of blocks of type `chunk`. These are either `Doc` blocks that contain documentation, or `Code` blocks that contain the chunk header as a `string array` and the source code.

Code chunk 5: «lepton.ml»

boilerplate

```
<<boilerplate>>
type chunk = Doc of string | Code of string array * string;;
```

We define the `option` type as well as two helper functions.

Code chunk 6: «lepton.ml (part 2)»

```
type option = {
  mutable part_number : int;
  mutable write : bool; mutable expand : bool;
  mutable chunk_format: string; mutable output_format: string;
  mutable interpreter : string;
};;
let option_copy o = {o with part_number = 0};; (* independent copy of the object *)
let option_print name o =
  Printf.printf "%s (part %i%s%s):\t" name o.part_number (if o.expand then " expand" else "") (if o.write then " write" else "");
  Printf.printf "chunk as %s, " o.chunk_format;
  if o.interpreter <> "none" then Printf.printf "exec with %s, output as %s, " o.interpreter o.output_format;
  Printf.printf "\n%!";;
```

`make_get_item` is a generic storage function based on hash tables. `make_get_item` creates a hidden vault, initially populated with `initial` (key,value) pairs. `make_get_item` returns a function for accessing the elements in the vault. When the requested an item from the vault is not found, a new item is added via the `fnew` function.

Code chunk 7: «lepton.ml (part 3)»

```
let make_get_item initial fnew = let open Hashtbl in
  let storage = create 30 in List.iter (fun (key,value) -> add storage key value) initial;
  fun key -> try find storage key with Not_found -> (add storage key (fnew key); find storage key);;
```

Lepton uses vaults created by `make_get_item` for

- associating chunk names with the concatenated chunk contents. These are stored as extensible buffers; the lexical analyzer appends chunk contents to these buffers,

Code chunk 8: «lepton.ml (part 4)»

```
let get_chunk = make_get_item [] (fun (s:string) -> Buffer.create 100);;
```

- associating chunk names with their options. The function is itself hidden in the `parse_chunklabel` function defined in 10,
- associating chunk names with output channels when writing to disk,

Code chunk 9: «lepton.ml (part 5)»

```
let send_to_file = let get_file = make_get_item [] open_out in
  fun msg file_name -> let oc = (get_file file_name) in output_string oc msg; flush oc;;
```

- associating process names with a process (see chunk 11).

The following functions are responsible for transforming the chunk header into a value of type `option`. This happens in two stages. During lexical analysis, the chunk header is transformed into a `string array` by the `split_header` function. In particular, this extracts the chunk name as first element of this array.
The `parse_chunklabel` function is called during chunk interpretation. For each chunk, the option vault is queried for a previous chunk with the same name, or the current default options. These options are then modified and stored in the option vault according to the current chunk header. Global default options are implemented as the special chunk name `lepton_options` and can be modified with a chunk of that name.

Code chunk 10: «lepton.ml (part 6)»

```
let split_header = fun s -> Array.of_list (Str.split (Str.regexp "[ \t]+") s);;
let parse_chunklabel = (* Parse the chunk label into name, option structure *)
  let defaults = { part_number=0; write=false; expand=false; chunk_format="text"; output_format="hide"; interpreter="none"; } in
  let get_option = make_get_item [("lepton_options",defaults)] (fun _ -> option_copy defaults) in
  function args ->
    let o = get_option args.(0) in o.part_number <- o.part_number + 1;
    let option_spec =
      ("-write", Arg.Unit (fun _ -> o.write <- true) , "write chunk to disk") ::
      ("-nowrite", Arg.Unit (fun _ -> o.write <- false) , "do not write chunk to disk (default)") ::
      ("-expand", Arg.Unit (fun _ -> o.expand <- true) , "expand chunk in documentation") ::
      ("-noexpand", Arg.Unit (fun _ -> o.expand <- false) , "do not expand chunk in documentation (default)") ::
      ("-chunk", Arg.String (fun s -> o.chunk_format <- s), "chunk type for pretty-printing") ::
      ("-output", Arg.String (fun s -> o.output_format <- s), "output type for pretty-printing") ::
      ("-exec", Arg.String (fun s -> o.interpreter <- s; o.output_format <- "text"), "send chunk to external interpreter") :: [] in
    Arg.parse_argv ~current:(ref 0) args option_spec (fun _ -> ()) "Wrong option in chunk header.\nusage : "; option_print args.(0) o; o;;
```

The mechanism for external interpretation works as follows. The `send_to_interpreter` function hides a process vaults which contains functions that execute code and return the output as a string. When the process name is not found in the vault, a new process / instance must be created. The process name is matched by prefix to a list of process creators. We provide a `make_process_creator` function for convenience. It is documented in Section 2.4. Note that the function `make_get_item` is not used for the list of process creators because matching happens by prefix.

Code chunk 11: «`lepton.ml (part 7)`»

make_process_creator

```
let process_creators = ref [("none",fun () -> (fun (s:string) -> ""))];;
let register_process_creator name f = process_creators := (name,f) :: !process_creators;;
let send_to_interpreter = (* return output of external chunk interpretation *)
  let rec assoc_prefix name = function
    | (key,value)::_ when String.length name >= String.length key && key = String.sub name 0 (String.length key) -> value
    | a :: b -> assoc_prefix name b | [] -> failwith ("send_to_interpreter : cannot find " ^ name) in
  let get_process = make_get_item [] (fun process_name -> (assoc_prefix process_name !process_creators) ()) in
  fun msg process_name -> (get_process process_name) msg;;
<<make_process_creator>>
```

Finally, we define the following variables that are used for communicating between the main functions and the modules. In particular, `formatter` is populated at runtime depending on the command-line option.

Code chunk 12: «`lepton.ml (part 8)`»

```
let lepton_ic = ref stdin;;
let lepton_oc = ref stdout;;
let formatter = ref (fun (name:string) o (l:string list) (chunk:string) (output:string) -> ignore(o.write) );;
```

2.3 Lexing equals syntax

Specification

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 [5], 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 2.4 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.

The lexical analyzer is responsible for transforming a stream of characters (from the Lepton file) into a series of chunks. We first define variables and regular expressions.

Code chunk 13: «`lexer.mll`»

boilerplate

```
<<boilerplate>>
{ open Hashtbl;;  open Buffer;; open Lepton;;
  let accu = Queue.create ();; let buffer = create 100;;
}
let char = [^ '\n']
let blank = [' ' '\t']
let chunk_start = "<<" (char* as h) ">>=" char* "\n"? as s
let linput = "\\Li" "nput{" (char* as file) "}" char* "\n"? as s
let lexp = "\\L" "expr{" ([^ '}' '\n']* as process) "}{" ([^ '}' '\n']* as code) "}"
let chunk_ref = (blank* as b) "<<" (char* as h) ">>" char* "\n"? as s
```

The lexical analysis of a Lepton file is composed of two main rules. The `lexer` rule is used when lexing documentation. In this state, only the `linput` directive is recognized. The `chunk_start` lexeme opens a chunk block.
The `gobble` rule reads the character stream until the end of code chunk; it uses the integer variable `level` to track the nesting level of code chunks. This rule ignores `@@` at the beginning of a line in the first pass to preserve the nesting structure.

Lexing a file starts with the `shabang` rule that ignores the first line when it starts with `#!`, starts lexing in documentation mode, and adds any trailing content.

Code chunk 14: «lexer.mll (part 2)»

```
rule shabang = parse    | ("#!" char* "\n")? { lexer lexbuf; Queue.add (Doc (contents buffer)) accu; accu}
and lexer = parse
    | chunk_start { Queue.add (Doc (contents buffer)) accu; clear buffer;
                    let a = split_header h in ignore(gobble 1 lexbuf); add_buffer (get_chunk a.(0)) buffer;
                    Queue.add (Code (a,contents buffer)) accu; clear buffer; lexer lexbuf }
    | linput { lexer (Lexing.from_channel (open_in file)); lexer lexbuf }
    | char* "\n"? as s { add_string buffer s; lexer lexbuf } | eof {}
and gobble level = parse
    | chunk_start          { add_string buffer s; gobble (level+1) lexbuf }
    | "@@" char* "\n"?  as s { add_string buffer s; gobble level lexbuf }
    | "@"  char* "\n"?  as s { if (level > 1) then (add_string buffer s; gobble (level-1) lexbuf) else s;}
    |     char* "\n"?  as s { add_string buffer s; gobble level lexbuf }
    | eof                 { failwith "Lexing : eof not permitted in gobble mode";}
```

To keep all the syntax elements in the same file, we define the `lexpr` and `expand` rules that respectively interpret the contents of documentation chunks and code chunks. `lexpr` enables embedded code in documentation blocks and replaces `lexp` with the code output. `expand` performs recursive expansion of references in code chunks.
The `expand` rule uses two temporary strings for storing the plain version (with `@@` replaced by `@`) and the expanded versions of chunk contents. The `gobble` rule is used in `expand` to determine the limits of a nested chunk. Unlike in the documentation case, the chunk start and the chunk end lines must appear in `bplain` and `bexp`.

Code chunk 15: «lexer.mll (part 3)»

```
and lexpr = parse
    | lexp { output_string !lepton_oc (send_to_interpreter (code^"\n") process); lexpr lexbuf;}
    | _ as c { output_char !lepton_oc c; lexpr lexbuf; } | eof { flush !lepton_oc; }
and expand bplain bexp = parse
    | chunk_start  { clear buffer; let l = gobble 1 lexbuf in
                     expand (bplain^s^(contents buffer)^l) (bexp^s^(contents buffer)^l) lexbuf; }
    | chunk_ref { let h_contents = contents (get_chunk h) in
                  if String.length h_contents = 0 then Printf.printf "WARNING: ref <<%s>> is empty or missing.\n%!" h;
                  let _,expanded = expand "" "" (Lexing.from_string h_contents) in
                  let indented = String.concat "" (List.map (fun s -> b^s^"\n") (Str.split (Str.regexp_string "\n") expanded)) in
                  expand (bplain^s) (bexp^indented) lexbuf; }
    | "@@" (char* "\n"? as s) { expand (bplain^"@"^s) (bexp^"@"^s) lexbuf; }
    |      char* "\n"?  as s { expand (bplain^s) (bexp^s) lexbuf; } | eof {bplain,bexp}
and chunkref_list = parse
    | chunk_start { gobble 1 lexbuf; chunkref_list lexbuf; }
    | chunk_ref   { h :: chunkref_list lexbuf; }
    | char* "\n"? { chunkref_list lexbuf; } | eof { [] }
```

`lexer.mll` does not contain legitimate OCaml code. It must be processed by `ocamllex` to produce the actual lexer in the file `lexer.ml`(see Section 3). The actual lexer is implemented as a Deterministic Finite Automaton for efficiency.

N.B. Lexical analyzers can only take into account the past context; you need a syntax analyzer or parser to look at the downstream context.

2.4 External interpreters

Specification

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 2.5).

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.

The `interpreters.ml` file contains the definition of all recognized external interpreters. More precisely, it contains process creators, i.e. functions for creating a new instance of a given interpreter. Each process creator must be registered in the `process_creators` list (see 11).

Most process creators can be created by the following function. We first launch the new instance of the process with `open_process`, and retrieve the input and output channels. We then create a function that sends data to the input channel and reads from the output channel.

The main difficulty is that we let a process run in the background so that further instructions can be executed in the same environment. Consequently, the output channel is not closed, and reading from this channel results in a deadlock. To escape from this situation, and continue to interpret the Lepton file, we send a question to the interpreter after the chunk contents, and read from the output channel until we get the expected answer.

In the current implementation, reading from the output channel happens line by line. The answer is a string. Everything that follows the answer is ignored.

Code chunk 16: «make_process_creator»

```
let make_process_creator open_process question answer = fun () ->
  let (oc_in,oc_out) = open_process () and l = ref "" and b = Buffer.create 10 in
  let rexp_answer = Str.regexp ("\\(.*\\)"^answer) in
  fun msg -> Printf.fprintf oc_out "%s%s%!" msg question; (* Printf.printf "%s%s%!" msg question; *)
    Buffer.clear b; while (l := input_line oc_in;not (Str.string_match rexp_answer !l 0))
      do Buffer.add_string b (!l ^ "\n") done; Buffer.contents b ^ Str.matched_group 1 !l;;
```

The process creators for the UNIX shell, Python and R interpreters are easily defined. Note that most interpreters require a newline character to terminate the question.

Code chunk 17: «interpreters.ml»

boilerplate

```
<<boilerplate>>
open Lepton;;
let str = string_of_float (Random.float 1.);;
register_process_creator "shell" (make_process_creator (fun _ -> Unix.open_process "sh") ("echo \"" ^ str ^ "\"\n") str) ;;
register_process_creator "python" (make_process_creator (fun _ -> Unix.open_process "python -i") ("print \"" ^ str ^ "\"\n") str);;
register_process_creator "R" (make_process_creator (fun _ -> Unix.open_process "R --slave") ("cat(\"" ^ str ^ "\\n\")\n") str);;
```

The OCaml process creator is more complex. We must set the `TERM` environment variable to empty, otherwise the interpreter assumes a full-fledged UNIX terminal and outputs color codes. Additionally, we suppress the first two lines that correspond to the OCaml version.

Code chunk 18: «interpreters.ml (part 2)»

```
let ocaml_creator =
  let open_proc = fun _ ->
    (let oc_in,oc_out,oc_err = Unix.open_process_full "ocaml -noprompt" [|"TERM="|] in
    ignore(input_line oc_in); ignore(input_line oc_in); oc_in, oc_out ) in
  make_process_creator open_proc ("print_float " ^ str ^ ";;\n") (str^"- : unit = ()")
in register_process_creator "ocaml" ocaml_creator;;
```

This is the Scilab interpreter. Communication with Scilab requires non-blocking pipes.

Code chunk 19: «interpreters.ml (part 3)»

```
let scilab_open = fun _ ->
  let entrypipe_r, entrypipe_w = Unix.pipe() and exitpipe_r, exitpipe_w = Unix.pipe() in Unix.set_nonblock entrypipe_r;
  let oc_in = Unix.in_channel_of_descr exitpipe_r and oc_out = Unix.out_channel_of_descr entrypipe_w in
  ignore(Unix.create_process_env "scilab-cli" [| "scilab-cli" |] [|"SCIHOME=/tmp"|] entrypipe_r exitpipe_w exitpipe_w);
  oc_in,oc_out;;
register_process_creator "scilab" (make_process_creator scilab_open ("disp(\"325\");\n") ("325"));;
```

2.5 Documentation formatters

Specification

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.

Note that current formatters are not aware of nesting, and may insert incorrect links and references.

Code chunk 20: «formatters.ml»
boilerplate

```
<<boilerplate>>
open Printf;; open Lepton;;
let tex = let r = Str.regexp_string "_" in fun s -> Str.global_replace r "\\_" s ;;
let send_to_latex_minted = fun name o reflist chunk output ->
  let plain s = fprintf !lepton_oc "%s%!" s
  and leptonchunk s = function | "hide" -> () | "verb" -> fprintf !lepton_oc "%s%!" s;
    | format -> fprintf !lepton_oc "\\b\101gin{minted}[frame=single,fontsize=\\footnotesize]{%s}\n%s\\\101nd{minted}\n%!" format s
  and leptonfloat_begin () =
    fprintf !lepton_oc "\\b\101gin{leptonfloat}\n\\caption{%s%s}\\label{%s}\n%!"
      (tex name) (if o.part_number = 1 then "" else sprintf " (part %i)" o.part_number)
      (if o.part_number = 1 then name else name ^ string_of_int o.part_number);
    fprintf !lepton_oc "\\vspace*{\\leptonlb}\\footnotesize{\\texttt{%s}}\\vspace*{-\\leptonlc}\n%!"
      (String.concat "\\, " (List.map (fun d -> Printf.sprintf "\\index{%s}\\hyperref[%s]{%s}" (tex d) d (tex d)) reflist));
  and leptonfloat_end () = fprintf !lepton_oc "\\\101nd{leptonfloat}\n%!"; in
  match o.chunk_format with
    | "hide" | "verb" -> if o.chunk_format = "verb" then plain chunk;
      if o.interpreter <> "none" then leptonchunk output o.output_format
    | f1 -> leptonfloat_begin (); leptonchunk chunk f1;
      match o.interpreter,o.output_format with
        | "none",_ | _, "hide" -> leptonfloat_end ();
        | _, "verb" -> leptonfloat_end (); plain output;
        | _, f2 -> plain ("\\vspace*{-\\leptonld}Interpret with \\texttt{" ^ tex o.interpreter ^ "}\\vspace*{-\\leptonle}\n");
          leptonchunk output f2; leptonfloat_end ();
;;
```

Here is an example of a chunk formatted by `latex_minted`.

```
\begin{leptonfloat}
\caption{clean}\label{clean}
\vspace*{\leptonlb}\footnotesize{\texttt{}}\vspace*{-\leptonlc}
\begin{minted}[frame=single,fontsize=\footnotesize]{sh}
# Remove pre-existing files, output is empty on success
rm *.ml *.bin *.pdf *.mll *.cmi *.cmo
rm lepton_manual.* hello.*
\end{minted}
\vspace*{-\leptonld}Interpret with \texttt{shell}\vspace*{-\leptonle}
\begin{minted}[frame=single,fontsize=\footnotesize]{text}
\end{minted}
\end{leptonfloat}
```

Code chunk 21: «formatters.ml (part 2)»

```
let substitute_split s rexp ftext fdelim = let open Str in
  String.concat ""
    (List.map (function | Text m -> ftext m | Delim m -> ignore (string_match rexp m 0); fdelim m) (full_split rexp s))
;;
let rexp_ref = Str.regexp "^\\([ \t]*\\)<\060\\(.*\\)>>\n";;
let send_to_tex name o reflist chunk output = (* echo to documentation, in plain TeX format *)
  let plain s = Printf.fprintf !lepton_oc "%s%!" s
  and leptonchunk s = function | "hide" -> () | "verb" -> Printf.fprintf !lepton_oc "%s%!" s;
    | format -> Printf.fprintf !lepton_oc "\\b\101gin{verbatim}\n%s\\\101nd{verbatim}\n%!" s
  and leptonfloat_begin () =
    Printf.fprintf !lepton_oc "\\b\101gin{leptonfloat}\n\\caption{%s%s}\n\\label{%s}"
      (tex name) (if o.part_number = 1 then "" else Printf.sprintf " (part %i)" o.part_number)
      (if o.part_number = 1 then name else name ^ string_of_int o.part_number);
    Printf.fprintf !lepton_oc "\\vspace*{\\leptonlb}\\footnotesize{\\texttt{%s}}\\vspace*{-\\leptonlc}\n"
      (substitute_split chunk rexp_ref (fun _ -> "") (fun d0 -> let d = Str.matched_group 2 d0 in Printf.sprintf "\\index{%s}\\hyperref[%s
  and leptonfloat_end () = Printf.fprintf !lepton_oc "\\\101nd{leptonfloat}\n%!"; in
  match o.chunk_format with
    | "hide" | "verb" -> if o.chunk_format = "verb" then plain chunk;
      if o.interpreter <> "none" then leptonchunk output o.output_format
    | f1 -> leptonfloat_begin (); leptonchunk chunk f1;
      match o.interpreter,o.output_format with
        | "none",_ | _, "hide" -> leptonfloat_end ();
        | _, "verb" -> leptonfloat_end (); plain output;
        | _, f2 -> plain ("\\vspace*{-\\leptonld}Interpret with \\texttt{" ^ tex o.interpreter ^ "}\\vspace*{-\\leptonle}\n");
          leptonchunk output f2; leptonfloat_end ();
;;
```

Code chunk 22: «formatters.ml (part 3)»

```
let send_to_html name o reflist chunk output =
  begin match o.chunk_format with
    | "hide" -> ()
    | _ -> Printf.fprintf !lepton_oc "\n<pre id=leptonchunk>\n%s</pre>\n%!" chunk ; end;
  begin match o.output_format with
    | "hide" -> ()
    | _ -> Printf.fprintf !lepton_oc "\n<pre id=leptonoutput>\n%s</pre>\n%!" output ; end;
;;
```

Code chunk 23: «formatters.ml (part 4)»

```
let send_to_creole name o reflist chunk output =
  begin match o.chunk_format with
    | "hide" -> ()
    | _ -> Printf.fprintf !lepton_oc "\n{{{\n%s}}}\n%!" chunk ; end;
  begin match o.output_format with
    | "hide" -> ()
    | _ -> Printf.fprintf !lepton_oc "\n{{{\n%s}}}\n%!" output ; end;
;;
```

The default formatter is set to `latex_minted`. The `set` function is used for parsing the command-line.

Code chunk 24: «formatters.ml (part 5)»

```
Lepton.formatter := send_to_latex_minted;;
let set = function
  | "latex_minted" -> Lepton.formatter := send_to_latex_minted
  | "tex"          -> Lepton.formatter := send_to_tex
  | "creole"       -> Lepton.formatter := send_to_creole
  | "html"         -> Lepton.formatter := send_to_html
  | _ -> failwith "unknown selected formatter";;
```

3 Obtaining the software

Specification

Lepton can be downloaded from Zenodo [4], 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.

Everything in the Lepton distribution is intended to be generated from the `lepton.nw` file by the Lepton executable. To initiate this bootstrapping process, the archive contains OCaml source files as well as the following script to compile them. This will produce a standalone binary. A standard installation of Ocaml is sufficient for this step. Lepton is written using only the standard library.

Code chunk 25: «make.sh»

```
ocamllex lexer.mll
ocamlopt -o lepton.bin str.cmxa unix.cmxa lepton.ml lexer.ml interpreters.ml formatters.ml main.ml
```

Interpret with `shell`

```
101 states, 393 transitions, table size 2178 bytes
1974 additional bytes used for bindings
```

With an existing Lepton executable, the command `lepton lepton.nw -o lepton.tex` will:

1. extract the source files from the code chunks, and deal with chunk references (such as `make_process_creator` 11),
2. run the compilation commands in `make.sh` to produce an executable
3. run the PDF compilation commands in Section 4 to produce the `lepton_manual.pdf` and the tutorial. N.B. this step makes a call to the `lepton` executable which needs to be in the `PATH` variable.

The rendered `lepton.pdf` is the intended way to read the Lepton source code. It can be generated by LaTeX with the following commands. You will need the `Pygments` Python library and the accompanying `minted` LaTeX package for syntax highlighting. These commands cannot be run by Lepton because `lepton.tex` is generated by Lepton.

Code chunk 26: «shell_not_executed»

```
xelatex -shell-escape -8bit lepton.tex
bibtex lepton.aux
xelatex -shell-escape -8bit lepton.tex
xelatex -shell-escape -8bit lepton.tex # LaTeX needs to execute twice to resolve references
```

4 The Lepton manual

4.1 Manual structure

The manual is embedded in this document in such a way that

- the program specifications appears near its implementation,
- the specification appears both inside this document and a standalone manual.
- only one instance appears in the `lepton.nw` file,

To that end, the manual is in fact composed of a series of chunks that describe the specification in LaTeX syntax. Each of these fragments appears in the `lepton.nw` source file next to the corresponding implementation. The output is constructed by assembling the corresponding chunks:

- the specification appears by using the chunk option `-chunk verb` for direct inclusion in the LaTeX output.
- we produce a standalone manual by writing a complete LaTeX file. (see below)

Let us first define the LaTeX header.

Code chunk 27: «lepton_manual.nw»

```
\documentclass[a4paper,10pt]{scrartcl}
\usepackage[T1]{fontenc}
\usepackage{lmodern}
\usepackage[english]{babel}
\usepackage{graphicx}
\usepackage[bindingoffset=0cm,width=19cm,height=28cm]{geometry}
\usepackage[bibliography=normal]{savetrees}
\usepackage[numbers]{natbib}
\setlength{\bibsep}{0pt plus 0.3ex}

\usepackage{float}
\usepackage{hyperref}

\newlength{\leptonla} \newlength{\leptonlb} \newlength{\leptonlc}
\newlength{\leptonld} \newlength{\leptonle} \newlength{\leptonlf}
\setlength{\leptonla}{2ex} \setlength{\leptonlb}{1ex} \setlength{\leptonlc}{1ex}
\setlength{\leptonld}{1ex} \setlength{\leptonle}{1ex} \setlength{\leptonlf}{2ex}
\newcommand{\lepton}{Lepton}
\makeatletter
\renewcommand\floatc@ruled[2]{{\bfseries #1:} \texttt{\flqq\relax#2\frqq}\par}
\renewcommand\fs@plaintop{\fs@plain
  \let\@fs@capt\floatc@ruled
  \def\@fs@pre{\relax\vspace*{-\leptonla}}%
  \def\@fs@mid{\relax}%
  \def\@fs@post{\relax\vspace*{-\leptonlf}}%
  \let\@fs@iftopcapt\iftrue}
\floatname{leptonfloat}{Code chunk}
\makeatother
\floatstyle{plaintop}
\newfloat{leptonfloat}{H}{lol}
\begin{document}

\section{The \lepton\ manual}
\label{sec:manual}
```

We provide a tutorial as the first section of the manual.

Code chunk 28: «lepton_manual.nw (part 2)»

```
\subsection{Tutorial}
\label{sec:tutorial}

\begin{minipage}{0.7\linewidth}

To write a ``hello world'' manuscript, we use the \verb hello.nw  file as an example of the \LaTeX-like syntax:
<<hello.nw -write -chunk tex>>=
\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 \verb hello.nw  into a legitimate \LaTeX\ document \verb hello.tex .
When processing a file, \lepton\ outputs the name of each encountered code snippet
and how it deals with it.

<<hello.tex -exec shell>>=
lepton -format_with tex hello.nw -o hello.tex
@

The \verb hello.tex  file is compiled with \verb pdflatex . The resulting
PDF file is displayed on the right.
<<hello.pdf -exec shell>>=
pdflatex -interaction=batchmode hello.tex
@
\end{minipage}
\begin{minipage}{0.3\linewidth}
  \centering
  \fbox{  \includegraphics[width=\linewidth]{hello.pdf}}
\end{minipage}
```

We then assemble the specification chunks from Section 2.

Code chunk 29: «lepton_manual.nw (part 3)»

spec_commandline spec_syntax spec_interpretation spec_formatting spec_availability

```
\subsection{Usage and command-line options}
<<spec_commandline>>

\subsection{Syntax}
<<spec_syntax>>

\subsection{Interpretation of code snippets}
<<spec_interpretation>>

\subsection{Formatting}
<<spec_formatting>>

This is the \LaTeX\ code produced by the \verb tex  formatter from the \verb hello_world  chunk in the tutorial.
\vspace*{-2mm}
\begin{verbatim*}
<<tex_output -exec shell -chunk hide -output verb>>=
head hello.tex -n 20 | tail -n 14
@
\end{verbatim*}

\subsection{Current implementation and availability}
<<spec_availability>>

\bibliographystyle{plainnat}
\bibliography{biblio_lepton}
\end{document}
```

Code chunk 30: «compile_manual»

```
./lepton.bin -format_with tex lepton_manual.nw -o lepton_manual.tex
pdflatex lepton_manual.tex
bibtex lepton_manual.aux
pdflatex lepton_manual.tex
pdflatex lepton_manual.tex  # LaTeX needs to execute twice to resolve references
```

References

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

[2] Donald E. Knuth. Literate programming. *THE COMPUTER JOURNAL*, 27:97–111, 1984.

[3] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system (release 3.12): Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, July 2011. URL `http://caml.inria.fr/distrib/ocaml-3.12/ocaml-3.12-refman.pdf`.

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

[5] N. Ramsey. Literate programming simplified. *Software, IEEE*, 11(5):97–105, 1994.