L2T: a Literate Programming tool (that extracts code from program source files towards TEX)

L2T release 2004Apr08, Documentation Revision: 1.59 as of Date: 2003/02/22 17:08:39

Christian Queinnec
LIP6 — Université Pierre et Marie Curie
4, place Jussieu, 75252 Paris Cedex 05, France
Email: `Christian.Queinnec@lip6.fr`

Abstract

This report describes an utility program that allows easy insertions of program source snippets towards TEX files. The originality of L2T is that it extracts definitions from the files where they appear and wraps them appropriately within TEX macros for insertion into documentation files. L2T decorrelates writing documentation from programming: it is therefore possible to separately develop programs and documentations and to merge them at the end to produce up to date final documents. L2T also has some pretty-printing capabilities (for Scheme definitions only) to produce denotations full of greek letters. L2T has been used since 1991.

This documentation is a reference guide to L2T and, being itself processed by L2T, it provides some examples of use both in source and final form.

L2T *sources are gathered in a file named* ***`l2t*.tar.gz`*** *which can be anonymously* ***`ftp`****-ed with* ***`http://www-spi.lip6.fr/~queinnec/Programs/l2t*`****. A binary* `rpm` *package also exists for Linux, see the WWW page below.*

The most up-to-date information may be read from:

`http://www-spi.lip6.fr/~queinnec/WWW/l2t.html`

This document has been processed by L2T *as identified by the following header:*

```
LiSP2TeX (Gambit) 2004Apr08 [Christian.Queinnec@lip6.fr]
```

1 Bits of history

Thinking that it would ease the writing and testing of my next book, I wrote in 1985 a program called `LLroff` that allowed to mix Le-Lisp[1] and `troff`. A `LLroff` file was a regular file which can be directly loaded by Le-Lisp where `troff` parts appear located within Lisp comments. A Le-Lisp-based filter was able to convert a whole `LLroff` file into an `troff` file. During this process, `troff` parts were extracted, Le-Lisp code was typeset in typewriter font as well as Lisp tests were run. By the way, a lesson of this experiment was that program excerpts look better if they are reproduced exactly as they are written: they do not gain to be automatically pretty-printed.

Year 1986 marked the time when I shifted to TEX so I convert `LLroff` into `ProTEX` which generates at will `troff` or TEX code, a good way to be independent of formatters. Several teaching documents and part of a book were produced with it. I also used a Macintosh[2] at that time and introduced the following trick which is worth mentioning since when you write texts on Lisp dialects, you are likely to mention many times Lisp expressions: this should be very easy and must not involve weird sequences of keys. The § character was a directly accessible character on the french version of the Macintosh keyboard. One § means: read the next S-expression and print it, surrounded with appropriate TEX macros. Two §§ means: read the next S-expression and evaluate it but do not print the result. Of course three §§§ just produces a single §.

Some years of use showed the limitation of the mixing approach. It is difficult to maintain files containing at the same time a lot of code and and a lot of text. This makes debugging painful: you have to scroll hard to find the interesting parts. Conversely, `make` automatically recompiles files any time you modify the documentation and thus wastes computing power as well as it introduces annoying delays. Moreover under Emacs, you have to switch from text-mode to Lisp-mode forth and back.

The aim of a documentation is to describe programs. Fragments of programs are usually inserted in the documentation in specific places. As the description goes, programs are usually debugged, commented or even improved. The whole problem is therefore to insert up to date excerpts. When producing documents, one also wants to show examples of use: nothing is better than computed examples since they are automatically faithful and accurate. This approach is now common and the CAML manual [Wei90] and the Le-Lisp manual both use this technique. Anecdoctically, computed examples are perfect but you still have to harmonize your comments to the produced answers. If you program `fibonacci` instead of `factorial`, do not write that `(factorial 10)` is 89 ! Since 89 is only the accurate value of `(fibonacci 10)`. Such teleological "errors" are difficult to see!

I thus decided to separate programs and texts and to merge them thanks to `ProTEX` and a new directive named `fromfile`. The directive §§`(fromfile "`*`filename`*`" `*`name`*`)` took a *filename* containing Lisp code and a *name*, it then read the mentioned file and extracted from it the definition of *name*, inserting it textually (duly surrounded with TEX macros) in lieu of the initial directive.

Recently, I decided to rewrite this tool in portable Scheme, it is called now LiSP2TEX for "Literate Scheme Programming to TEX[3]". LiSP2TEX is a small stand-alone program, written in Scheme that includes a complete Scheme reader and a customizable pretty-printer. I have been using it since 1991 for all my papers containing denotations or Scheme excerpts.

LiSP2TEX belongs to the family of systems coined as "Literate Programming". Several programs already existed for Scheme and LATEX. Scheme`WEB` from John D. Ramsdell is such a system that takes a mixed file and produces two files out of it: the first only contains Scheme code while the other is a LATEX file with Scheme parts reproduced verbatim. Another is the excellent SLATEX from Dorai Sitaram which automagically makes various Scheme tokens to appear with user-selected fonts. A third one is $\mathcal{ST}$O$\mathcal{L}$ [KH92] which, similarly to Scheme`WEB`, uses the mixing approach.

Becoming a professor at University Paris 6 in 1996, I decided to open up LiSP2TEX to other programming languages and most prominently to C which I taught. Thierry Saura wrote a code extractor for C that was

[1] Le-Lisp is a trademark of INRIA.

[2] Macintosh is a trademark of Apple Computer Inc.

[3] As more and more programs are written, the art of naming them becomes more and more difficult. It is unlikely that such an abuse of case sensitiveness is already in use somewhere. To use such an incredible name, I `alias` it or bury it within Makefiles.

incorporated into LiSP2TEX. During year 2000, I set up a protocol for LiSP2TEX to call external parsers (not written in Scheme) so now it may extract whatever snippets from whatever text files (not only programming languages) provided an appropriate external parser exists. As you may realize, the name “LiSP2TEX” appears to be more and more badly chosen so I change it into the less connoted L2T.

Acknowledgments

Many thanks to Titou Durand for his bright suggestions that greatly improved L2T these last years.

2 Reference Manual

L2T is a filter that identifies the directives of figure 1. They look like TEX macros (with their leading backslash) but they are not TEX macros. They are only specific strings that are looked for by L2T and appropriately handled.

`\FromFile`	`\PrettyFromFile`
`\Eval`	`\ShowEvaluationOf`
`\Print`	`\PrettyPrint`
`\FromCFile`	`\InsertFile`
`\LoadFile`	`\ExtractFromFile`

Figure 1: Directives recognized by L2T

It is also possible to use L2T with TeXinfo. TeXinfo uses directives starting with `@` instead of backslash. A customization variable allows you to set up the initial character of the directives to recognize. Of course, you must also customize L2T to suit your needs. This customization variable and its default setting are:

```
;;; This is the letter by which directives start. It is then easy to
;;; modify it to become # as for TeXinfo.
(define *directive-sign* #\\)
```

Directives may be split into two different sets.

General directives The directives `\FromFile`, `\PrettyFromFile`, `\FromCFile`, `\InsertFile` and `\ExtractFromFile` take an S-expression as unique argument. That S-expression should be a list that contains what is required for the directive to run.

Some options may be required to customize the process of the directive. Options must be gathered in a list as the first term of the S-expression. Options appear as DSSSL keywords that is, symbols ending with a colon. An option is always followed by a value.

Scheme-related directives The directives `\Eval`, `\Print`, `\PrettyPrint`, `\ShowEvaluationOf` and `\LoadFile` all handle S-expressions i.e. Lisp expressions. They evaluate, print or pretty-print these S-expressions.

2.1 How to invoke L2T

`LiSP2TeX [-hlLuUvV=a] [-I`*`dir`*`] ...[`*`customization-file`*`] ...-- [`*`file`*`] ...`

L2Tis run with the `LiSP2TeX` command. The result is always produced on the standard output. Several options are possible and many can be combined as in `-vhU`.

`-a` Stops at first error (by default, produces a warning and continue).

`-h` Suppress the emission of a header identifying L2T. This header is otherwise emitted as first lines of the standard output. The actual header appears at the bottom of the first page of this very document.

`-I`*`directory`* Augment the path to search files. More than one `-I` option may be specified. Observe that there is no space between `-I` and the directory name. By default, the path only contains the current directory.

`-L` Symbols are read in lowercase.

`-U` Symbols are read in uppercase.

`-=` Symbols are read and printed case-sensitively. This option is useful to reset options `-U`, `-u`, `-L` and `-l`.

`-l` Symbols are printed in lowercase.

`-u` Symbols are printed in uppercase.

`-v` Verbose mode. Progress is shown on the standard error output.

`-V` Highly verbose mode. More things are shown on the standard error output. `-V` implies `-v`. Some hints are given on the standard error stream while L2T runs. Errors are mentioned as well as files that are subject to the directives. Read errors are reported as precisely as possible with the line numbers. Evaluation errors are also handled and displayed with the line numbers.

`--version` Displays the identification of the current version of L2T on the standard error stream. This is actually "`2004Apr08`".

`--` Separates the files that customize the pretty-printer from the files that are to be filtered by L2T. If no customization file is present then the `--` option is unnecessary.

file Specify a file to be processed by L2T. Files are processed from left to right. Files are searched in the path that can be augmented with the `-I` option. Files appearing before the `--` option are customization files, files appearing after the `--` option are files to be processed. When a file is processed, its directory is temporarily added to the path (this makes easy local references from file to file within the same directory). If no files are mentioned then L2Tfilters its input stream.

The L2T program returns an exit code that is 0 if no problem was detected and not 0 in case of errors. When the `-a` option is mentioned and an error is detected, the returned exit status is 1.

L2T does not impose any extension for the files though *.l2t* is encouraged for customization files and *.bk* for files to be processed i.e. converted into a *.tex* file.

2.2 Directives

This section details all the directives of L2T. Variables to customize L2T are also commented.

`\Print` *S-expression*

The `\Print` directive allows to "cite" Scheme (or Lisp) code within a TEX paragraph. The `\Print` directive ensures the user to emit correctly parenthetized expressions. It reads the regular S-expression that follows and prints it as the argument of a TEX macro named `\snip[scheme]` which is defined (in the `LiSP2TeX.sty` file) to typeset its argument in `typewriter` font. The result is also set in a TEX group i.e. wrapped within { and }.

The S-expression which is read can be written over several lines and the result can also occupy several lines provided the definition of the `\snip[scheme]` TEXmacro allows it. The actual definition of `\snip[scheme]` which appears in the `LiSP2TeX.sty` file allows it.

Suppose you wrote "`\Print ( foo ( Bar ) )`", then L2T will expand it into the following TEX output: "`{\snip[scheme]{(foo (Bar))}}`" which in turn looks like `(foo (Bar))`.

Remarks, pitfalls and advices

Note that the S-expression is read then printed by Scheme so the read and written S-expression may appear differently. For example, the excerpt "`\Print #b10`" will be expanded as "`{\snip[scheme]{2}}`", provided the default output base is ten. The `\Print` directive is somewhat delicate to use on a symbol in a textual context. A symbol is ended by a delimiter in Scheme i.e. a space, an end of line, a parenthesis or a comment. These delimiters do not include the textual delimiters such as dot, comma, marks etc. So when you write "`\Print foo, Bar`" you get "`foo,` Bar" with the comma swallowed by the symbol.

L2T uses its own reader. This reader does not fully respect the IEEE Scheme Standard [IEE91]: the syntax of symbols is expanded since vertical bars and backslashes are allowed within symbols as in COMMON LISP [Ste90]. Conversely, numbers are restricted to signed integers or floating point numbers written with a single dot (moreover the dot must always be preceded by at least one figure). Tabulations and carriage returns are considered as whitespaces.

L2T uses its own printer. This printer prints quotations and quasiquotations in abbreviated format i.e. with quotes, backquotes and commas.

Customization

The reader is controlled by the following variables which rule how symbols are read. If `*respect-input*` is true then symbols are read case-sensitively, otherwise if `*uppercase-input*` is true, symbols are read in uppercase otherwise if `*lowercase-input*` is true they are read in lowercase.

```
;;; Defines case sensitivity.
(define *respect-input* #t)
(define *uppercase-input* #f)
(define *lowercase-input* #f)
```

The printer is controlled by the following variables which rule how symbols are printed. If `*respect-output*` is true then symbols are printed as they are read, otherwise if `*uppercase-output*` is true they are printed in uppercase otherwise if `*lowercase-output*` is true they are printed in lowercase. Moreover `*output-base*` specifies the base to print integers.

```
;;; By default respect case sensitivity.
```

```
(define *respect-output* #t)
(define *uppercase-output* #f)
(define *lowercase-output* #f)

;;; The output base for numbers
(define *output-base* 10)
```

L2T also uses the variable `*lisp-excerpt*`, the value of which is a format string that controls how is emitted the read S-expression.

```
(define *lisp-excerpt* "{\\snip[scheme]{~A}}")
```

\Eval *S-expression*

The `\Eval` directive reads and evaluates, for its effect, the following S-expression. It does not print the obtained result. This is useful to modify global customization variables, to perform tests etc. [See `\LoadFile` to load files.] The evaluation is performed by the underlying Scheme, so the S-expression is read with the preferred case-sensibility of the underlying Scheme.

The `\Eval` directive is useful to insert examples of computations: you just have to write something like "`\Print` π returns `\Eval (display` π`)`" or alternatively "`\Eval (set! test '`π`) \Eval (display test)` returns `\Eval (display (eval test))`". For instance, writing "`\Eval (display (+ 2 4))`" yields 6. See also the `\ShowEvaluationOf` directive.

Advanced examples

More intriguing effects can be obtained with `\Eval`, for example, you can define a function. Actually a function was defined between the previous word "function" and the colon that follows it. Now the directive "`\Eval(fact 10)`" yields 3628800. You can also trace the function and obtain:

```
(fact 4) <-
 (fact 3) <-
  (fact 2) <-
   (fact 1) <-
   (fact 1) -> 1
  (fact 2) -> 2
 (fact 3) -> 6
(fact 4) -> 24
```

Pitfalls

If you want the result of an `\Eval` directive to appear on the output stream, you can alternatively write `\Eval (display` *S-expression*`)`; the default output port is the standard output stream. This is slightly incorrect due to a problem of case sensitivity as specified by the `-UulL=` option. When L2T reads a symbol such as `Foo`, the generated symbol can, respectively, be `FOO`, `foo` or `Foo`. For instance, Scheme->C systematically converts to upper case and only recognizes keywords of special form in upper case. That is why S-expression for `\Eval` are read with the preferred case sensitivity of the underlying Scheme while S-expression for `\Print` are read with the case sensitivity set up by the user. This choice should, most of the time, save you from the burden of case awareness.

`\ShowEvaluationOf` *S-expression*

This directive combines the `\Print` and `\Eval` directives. It is often the case that one has to write "the expression `(foo ...)` yields `(value...)`". This can be simply typed as: "the expression `\ShowEvaluationOf (foo ...)`". The `\ShowEvaluationOf` directive reads an S-expression, echoes it, evaluates it and prints its result prefixed by the word "gives". For example, one can write "`\ShowEvaluationOf (+ 3 4)`" and simply obtain "`(+ 3 4)` gives 7".

Remark

If the expression prints some values during its evaluation, these are displayed after the expression is echoed but before the result is printed. Since this directive uses the underlying Scheme system, the S-expression is read with the case-sensitivity of the underlying Scheme rather than the user's sensitivity. You can partially compensate this effect after setting the output case to lowercase.

Customization

Several formats control the `\ShowEvaluationOf` directive. The first one is used to echo the read S-expression, the second one is used to display the result unless the result satisfies the `ignore?` predicate in which case the third format is used. This "hack" is used if you want to avoid printing the "non specified" value (`#<void>` in Gambit for instance). The `ignore?` predicate is pre-defined as:

```
(define (ignore? v)
  (memq v values-to-ignore) )
;;; In Gambit, define #¡void¿ to be ignored:
(define values-to-ignore
  (list (if #f 'ignore)) )
```

If the expression to evaluate yields an error, a message is produced to report this error; the fourth format is then used. When this format is used it receives the number of the line where the expression starts, the number of the line where it ends, the expression that was evaluated and the error message. The error message has no portable shape: it may be a string, a list etc.

```
(define *showevaluationof-input*    "\\snip[scheme]{~S} ")
(define *showevaluationof-output*   " gives \\snip[scheme]{~S}")
(define *showevaluationof-ignore-output* "~*")
(define *showevaluationof-error* "
%CAUTION: error lines ~A-~A on ~A
    yields ~A
%END OF ERROR
")
```

```
\FromFile (filename names...)
 \FromFile (options filename names...)
```

The `\FromFile` directive allows to retrieve definitions from a given file and to insert them in lieu of the directive, surrounded by the TEX macros `\snippet[scheme]` and `\endsnippet`. The `\FromFile` directive reads the S-expression that follows the directive. This S-expression must be a list which first term is the name of a file (a string); the other terms name the definitions that must be extracted. Alternatively, the filename may be prefixed with a list of options, see below the discussion of the possible "options".

If the filename is `""` then the last filename used by `\FromFile` or `\PrettyFromFile` is used. An error is signalled if there is no previously mentioned file. A directive that mentions a filename without any following names just sets up the default file.

The file is searched within the directories which appear in the `*path*` variable. An error is signalled if the file cannot be found. The `*path*` variable can be augmented via the `-I` option of the L2T command.

The mentioned file is read (only once) and each non atomic form is considered to be a definition. In a definition, the first symbol, the definer, is generally a symbol the name of which begins with the letters `d`, `e` and `f`. The second symbol (which can appear at various parenthetical level) is the name to be defined. This is naïve but works well with the various forms of `define`, `define-macro` etc.

Definitions are considered to start with a left parenthesis and end with the corresponding right parenthesis. When a definition is extracted, it is inserted in the TEX output exactly as it is written, all internal comments are kept as they were written.

To ease indexing definitions, a citation is also generated just before the `\snippet[scheme]` token, the defined name appears as the second argument of the TEX macro `\snippetCite`, the first one being the defining form. This TEX macro can be used to automatically index the definitions of a text.

Two methods exist to retrieve definitions.

- You may specify a single symbol. In which case the <u>first</u> definition that defines this symbol is retrieved. This is the usual case.
- You may specify a list i.e. a pattern. In that case, <u>all</u> definitions from the given file which are matched by this pattern are retrieved. This is particularly useful with macros like `define-method` [Que91] which may appear multiple times. This is also useful to retrieve a particular definition unreachable by the first method.

  A pattern can use the three special symbols `?-`, `...` and `-or-` to match whatever object or whatever sequence of objects or a disjunction of patterns. The second pattern can only appear within a list. All other symbols are only matched by themselves. Observe that `(...)` is not equivalent to `?-` since the former only accepts pure lists while the latter accepts anything.

If you specify more than one name to the `\FromFile` directive then only one section `\snippet[scheme]` ... `\endsnippet` is emitted but as many citations as there are names. You can of course specify a same name more than once.

Concerning the global comments that precede a definition, L2T can either output them as they are or eliminate all of them but the last block of them. Several customization variables exist for these effects. Moreover, it is possible to dissociate the emission of the preceding global comment from the emission of the definition itself: there again see the customization variables.

Examples

The file `commands.scm` is one of the source files of L2T. If you write "`\FromFile ("commands.scm" run-Print)`", you obtain

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Print
;;; Read and print a Sexpression. By default (see port-XXX.scm) try to
;;; respect input case.
(define (run-Print stream)
  (command-Print (numbering-read stream)) )
```

The exact expansion shows the citation which, by default in the `LiSPTeX.sty` file, does nothing. It is:

```
\snippetCite{define}{run-Print}
\snippet[scheme]
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Print
;;; Read and print a Sexpression. By default (see port-XXX.scm) try to
;;; respect input case.
(define (run-Print stream)
  (command-Print (numbering-read stream)) ) \endsnippet
```

To illustrate pattern matching, let us write `\FromFile ("" (define (?- fmt . ?-) ...))` to retrieve all definitions with a first variable named `fmt`. If we apply it on the proper sources of L2T, we obtain:

```
;;; Reports some progress
(define (command-echo fmt . args)
  (when *verbose*
    (LiSP2TeX-apply pp-format stderr-port fmt args)
    (flush-all-buffers) ) )
;;; reports some error on both streams (to see it in the output TeX)
(define (command-error fmt . arguments)
  (set! *exit-status* 1)
  (display-error fmt arguments)
  (flush-all-buffers)
  (if *stop-at-first-error*
      (begin
        (display-error *abort-format* arguments)
        (*exit*) ) ) )
;;; Default behavior for display-error.
(define (display-error fmt arguments)
  (LiSP2TeX-apply pp-format stdout-port fmt arguments)
  (LiSP2TeX-apply pp-format stderr-port fmt arguments) )
```

More precisely, we obtain:

```
\snippetCite{define}{command-echo}
\snippetCite{define}{command-error}
\snippetCite{define}{display-error}
\snippet[scheme]
;;; Reports some progress
(define (command-echo fmt . args)
  (when *verbose*
    (LiSP2TeX-apply pp-format stderr-port fmt args)
    (flush-all-buffers) ) )
;;; reports some error on both streams (to see it in the output TeX)
(define (command-error fmt . arguments)
  (set! *exit-status* 1)
  (display-error fmt arguments)
  (flush-all-buffers)
  (if *stop-at-first-error*
      (begin
        (display-error *abort-format* arguments)
        (*exit*) ) ) )
;;; Default behavior for display-error.
(define (display-error fmt arguments)
  (LiSP2TeX-apply pp-format stdout-port fmt arguments)
  (LiSP2TeX-apply pp-format stderr-port fmt arguments) ) \endsnippet
```

Customization

The output of the `\FromFile` directive is controlled through some variables. The variable `*lisp-citation*` defines how citation are emitted. The variable `*beginlisp-block*` is emitted before the insertion of the retrieved definitions. The variable `*interlisp-block*` is emitted between definitions. The variable `*endlisp-block*` defines how a serie of insertions is closed. All these variables are simple strings except `*lisp-citation*` which is a format string taking two arguments.

Concerning preceding global comments, the `*include-last-preceding-comment*` makes only visible the last block of comments that precede the extracted definition (leading spaces are omitted) [See below for an example]. Otherwise all the global comments that precede the definition are output.

```
(define *lisp-citation*   "\\snippetCite{~U}{~U}~%")
(define *beginlisp-block* "\\snippet[scheme]~%")
(define *interlisp-block* " ~%")
(define *endlisp-block*   " \\endsnippet ")
;;; This variable controls whether the last comment that precedes an
;;; expression should be extracted as well. [This comment is not
;;; separated from the following definition by an empty-line.]
(define *include-last-preceding-comment* #t)
```

Pattern matching is controlled by two global variables that defines the joker and the sequence of jokers. These are:

```
;;; This naive pattern matcher recognizes ?- and ... at any level.
;;; ?- matches any Sexp while ... matches any sequence of Sexp. This
;;; latter can only be used in the context of a list.
```

```
;;; CommonLisp does not accept ... as a valid symbol so use this periphrase;
;;; Users of the pseudo port would probably redefine ... to be an other thing
;;; for instance ??- or ???.
;;; (-or- pattern pattern .. pattern) matches any expression matching one of
;;; these patterns.
(define *mono-Sexp-pattern* '?-)
(define *poly-Sexp-pattern* (string->symbol "..."))
```

Extraction of comments is further customized by the following variables. The value of these variables should be functions mapping strings into strings. They respectively receive the preceding global comments and the definition (as strings), may perform whatever transformation they want, their result will be emitted with the `*lisp-appearance*` format.

```
(define *lisp-appearance* "~U~U")
;;; Filter the definition.
(define *filter-lisp-definition* identity)
;;; filter the global comments.
(define (default-filter-lisp-comment comment)
  (if *include-last-preceding-comment*
      (keep-last-lisp-comment comment)
      comment ) )
(define *filter-lisp-comment* default-filter-lisp-comment)
```

Note that the predefined function `keep-last-comment`, takes a string containing all the global comments that precede a definition and just return the last block of comments without trailing empty lines. You may change the `*lisp-appearance*` to insert back an empty line if you wish.

If you want to only extract the global comments that precede a definition, just change the `*lisp-appearance*` format to `"~U~*"` (see page 22 the definition of the "star" format). You may alternatively assign `*filter-definition*` to `(lambda (comment) "")`.

2.2.1 Options

A new implementation of `\FromFile` exists that is triggered when the `line-number:` option appears in the argument of `\FromFile` associated to a true value. In this case, the generation is customized differently with some additional information allowing to generate the line numbers of the extracted snippet (this requires the support of the snippet style (see *snippet.sty*).

The names of the customization variables are almost the same for the same meanings the old version of `\FromFile` had. Before a snippet is emitted, the `*citation*` format is used to generate the citation index and the line numbers, then `*begin-block*` is emitted. Every two extracted snippets are separated with `*inter-block*`. At the end, is emitted `*end-block*`. The same functions to filter the prepended comments and the body of the definition are applied.

```
(define *citation*
  "\\snippetCite{~A}{~A}~%\\setSnippetLineNumber{~U}{~A,~A,~A}~%")
(define *begin-block* "\\snippet[~U]\\snippetLineNumber~U~%")
(define *inter-block* " \\snippetLineNumber~U~%")
(define *end-block*   " \\endsnippet ")
(define *appearance* "~U~U")
(define *filter-lisp-comment* default-filter-lisp-comment)
;;; Filter the definition.
(define *filter-lisp-definition* identity)
```

Here is the same example (nearly) using the `line-number:` option. The directive now looks like `\FromFile((line-number: #t) "commands.scm" *citation* *filter-lisp-definition*)` and the result is:

```
(define *citation*
  "\\snippetCite{~A}{~A}~%\\setSnippetLineNumber{~U}{~A,~A,~A}~%")
;;; Filter the definition.

(define *filter-lisp-definition* identity)

```

Another option, the `nth:` option, allows to extract a selected subset of definitions. For instance, a directive such as `\FromFile((nth: (2 0)) "file" (define (foobar ...) ...))` produces the third and first definitions of `foobar` extracted from the *file* file.

The `nth:` option may be followed by a single number or by a list of numbers. A single number is considered to be analog to the list containing that single number. The `nth:` option may only be used with patterns since only patterns extract a number of definitions (a symbol only extracts the first definition related to that symbol).

`\PrettyFromFile` (*filename names...*)

This directive bears the same kind of similarity that exists between `\Print` and `\FromFile` with respect to `\PrettyPrint`. Instead of being inserted in the output stream as it is, the extracted S-expression(s) is(are) pretty-printed with the current pretty-printer. See below the `\PrettyPrint` directive.

`\PrettyPrint` *S-expression*

This directive reads the next S-expression and pretty-prints it with the current pretty-printer. This pretty-printer can be customized via the style files mentioned before the `--` option.

Pretty-printing is, by default, similar to the `write` function. Still, by default, the output of the pretty-printer appears in a LaTeX tabbing environment and in mathematical mode. The pretty-printer can easily be extended to print specially some types of values, some forms or some symbols. It offers some disambiguating schemes not to confuse variables.

Customization

The pretty-printed S-expression is preceded by the value of `*pretty-print-prologue*` then followed by the value of `*pretty-print-epilogue*`. They introduce a tabbing environment and set up mathematical mode: this eases producing denotations with greek letters. The exact definition can be found in the `LiSP2TeX.sty` file.

```
;;; The TeX condition \DenotationInline{true,false} controls whether
;;; lines are broken or not. So a common idiom is:
;;;   \DenotationInlinetrue \PrettyPrint (..)
;;;   to obtain a pretty printed version of a function in a single line.
```

```
;;; See for more details LiSP2TeX.sty

(define *pretty-print-prologue* "%%%~%\\DenotedLisp~%")
(define *pretty-print-epilogue* " %%%~%\\EndDenotedLisp ")
```

Three TeX macros control how produced lines are broken. These macros only appear at the end of lines so it is simple to comment them by hand if needed. Due to the tabbing environment, there is a notion of current left margin. The `\newline` macro terminates the current line and goes to the next one to the current left margin position. A new left margin can be defined with `\setandincrindent` which will be active until a `\decrindent` macro is encountered.

The predefined ways to print definitions, alternatives etc. break lines as much as possible. Some denotations are small enough to appear on a single line so all the three previous macros can be activated or inactivated by the TeX macros `\DenotationInlinetrue` and `\DenotationInlinefalse`.

\FromCFile (*filename names...*)

This is an additional directive introduced by Thierry Saura; it allows to extract a C definition from a C file, that is, a file with `.c` or `.h` extensions. The definition is extracted from the specified file and is inserted in the output stream surrounded by the TeX macros `\snippet[c]` and `\endsnippet`. The mentioned names must be strings (since C is case-sensitive). A name specify which C definitions should be extracted based on the occurrence of that name in the defining part of the definitions. Definitions of functions, variables, macros and types may be extracted. The defining part of a C definition generally starts with the first meaningful character, ends with the final semi-colon at the end of the definition (function definitions may finish with a closing brace without any semi-colon) but excludes any brace-nested sequence of characters. This is not perfect but sufficient.

When a file is read as a C file with `\FromCFile`, it cannot be used with the other `\FromFile` and related directives. An error is signalled if there is no previously mentioned file. A directive that mentions a filename without following names just sets up the default file.

The file is searched within the directories which appear in the `*path*` variable. An error is signalled if the file cannot be found. The `*path*` variable can be augmented via the `-I` option of the L2T command.

Examples

The file `essai.c` is one of the source files of L2T. If you write "`\FromCFile ("essai.c" "another_type" "str1" "hux" "foobar")`", you obtain:

```
typedef struct another_type {
  int a;
  int b;
} *(another_type)();
char *str1 = "\nmat" "ching }";
#define hux(a,b) \
   ((a)          \
    + (b) )
static const int
foobar (int a,
        int b) {
  return hux(a-b);
}
```

Customization

The output of the `\FromCFile` directive is controlled through some variables. The variable `*beginc-block*` is emitted before the insertion of retrieved C definitions. The variable `*interc-block*` is emitted between each definition. The variable `*endc-block*` is emitted after the insertion of a series of C definitions.

Two additional variables control how C definitions appear. The variable `*double-backslash*` if true doubles all emitted backslashes. This is quite convenient when a string contains a `\n` for instance. The `*omit-comments*` variable if true skips over all the inner comments of the definition.

These are the default values of these variables.

```
(define *beginc-block* "\\snippet[c]~%")
(define *interc-block* " ~%")
(define *endc-block*   " \\endsnippet ")
;;; Extract a substring but, It is possible to ask to double backslah
;;; for TeX, and leave C comments as they are.

(define *double-backslash* #f)
(define *omit-comments* #f)
```

Caveats

The `\FromCFile` directive roughly behaves as the `\FromFile` directive except that files are supposed to be written in C rather than in Lisp; no pattern matching is possible for extraction; all retrieved definitions are emitted rather than the first one. Therefore you must be precise with the names you use to avoid extracting multiple matching definitions.

The `\FromCFile` directive may probably applied on non-C files provided they use braces, quotes and semi-colons as C does. L2Tdoes not implement the exact C grammar but only a reasonable(?) approximation. Last word, functions defined with the old C-style (often qualified as K&R style) cannot be extracted.

`\InsertFile (`*filename* `...)`

The `\InsertFile` directive outputs the content of the specified files. The content of the files may be prefixed and suffixed as defined by the `*insertfile-format*` customization variable. This format outputs the name of the file (in a TeX comment), the name of the snippet environment to use and the content of the file. It may be customized to take care of a missing final newline among other things.

```
(define *insertfile-format*
  "% Inserted File ~A~%\\snippet[~U]~%~U\\endsnippet" )
```

The file is searched within the directories that appear in the `*path*` variable. An error is signalled if the file cannot be found. The `*path*` variable can be augmented via the `-I` option of the L2T command.

The snippet environment is computed after the extension of the inserted file. By default, this is `scheme` for *.scm* or *.l2t*, `c` for *.c* or *.h*. The snippet LaTeX style takes care of that name to typeset the program excerpt. It is possible to add or alter the associations between filename extensions and names of snippet environments. The `define-snippet-environment` function takes the name of a snippet environment and a number of extensions and add them. By default, L2T is defined with:

```
(define-snippet-environment "scheme" "scm" "l2t")
(define-snippet-environment "c"      "c" "h")
```

You may remove those predefined associations with `#f` as first argument to `define-snippet-environment`.

\LoadFile (*filename ...*)

This directive loads (i.e. evaluates the content of) the specified files. These files are searched within directories which appear in the `*path*` variable. An error is signalled if the file cannot be found. The `*path*` variable can be augmented via the `-I` option of the L2T command.

This directive is far better than the, now obsolete, old trick: `\Eval(load "filename")`.

\ExtractFromFile (*filename key ...*)

This directive generalizes the `\FromFile` and the `\FromCFile` directives (and shares most of its code with the new implementation of `\FromFile`). It allows to extract parts of any file of any type provided that the extension of that file is associated to an external parser. The external parser is any executable program that takes two arguments: the name of the file to parse, the name of the file where to store the results of the parsing. These parsing results are then used to extract parts of the original file.

The result file is a series of lines structured as follows (see example below):

(*start real-start end* "*name1*" ..."*nameN*")

Every part of the file to parse that may be extracted with L2T is represented by a line. The first three terms of the line are natural numbers: they are interpreted as positions within the parsed file. The first number tells where starts the named part, the second number tells where really starts the named part. The second number cannot be less than the first. The difference is the possible leading comment preceding the named part. The third number tells where ends the named parts.

The names that follow specify the fully qualified name of the part. Parts may be embedded (this is the case in Java or Ada with `class` and `method` or `package` and `function`).

For instance, here is the content of the file produced by *./LiSP2TeX-parser-perl.pl* for the *./LiSP2TeX-parser-perl.pl* file itself:

```
(595 653 836  "slurp_file")
(836 838 990  "report")
(906 907 926  "report" "$dummy")
(1088 1148 1526  "analyse_body")
(1309 1310 1348  "analyse_body" "$start")
(1348 1349 1394  "analyse_body" "$real_start")
(1394 1395 1437  "analyse_body" "$end")
(1437 1438 1460  "analyse_body" "$name")
```

To extract a piece of code whose name is N within a broader part named P within file `f.x`, just say:

`\ExtractFromFile("f.x" "P" "N")`

The extracted piece will be surrounded by `\snippet[x]` and `\endsnippet` where `x` is the snippet environment associated to the extension of the file.

The file from which to extract named parts is searched within the directories which appear in the `*path*` variable. An error is signalled if the file cannot be found. The `*path*` variable can be augmented via the `-I` option of the L2T command.

Initially no external parsers is defined. If you want to define one, you should use the `define-parser` function that takes the extension to recognize (without leading dot) and the name of the program to parse files. For instance, you may use that example to extract (very) simple snippets from a Perl file:

```
(define-parser "pl" "./LiSP2TeX-parser-perl.pl")
```

This command tells that files ending with `.pl` may be parsed with *./LiSP2TeX-parser-perl.pl*. Pieces of Perl code may be extracted within a `\snippet[perl]` ... `\endsnippet` environment. For instance, The following directive extracts a function and a variable within that function:

`\ExtractFromFile("LiSP2TeX-parser-perl.pl" "report" ("report" "$dummy"))`

The result is:

```
sub report {
    my ($start, $real_start, $end, $embed, $name) = @_;
    my $dummy = 42;
    print OUT "($start $real_start $end $embed \"$name\")\n";
}
    my $dummy = 42;
```

Customization

The `\ExtractFromFile` directive is driven by the same customization variables as the new implementation of `\FromFile` except that the filtering functions are `*filter-external-comment*` to filter the leading comments and `*filter-external-body*` to filter the definition itself.

Here are the default values for these customization variables:

```
(define *citation*
  "\\snippetCite{~A}{~A}~%\\setSnippetLineNumber{~U}{~A,~A,~A}~%")
(define *begin-block* "\\snippet[~U]\\snippetLineNumber~U~%")
(define *inter-block* " \\snippetLineNumber~U~%")
(define *end-block*   " \\endsnippet ")
(define *appearance* "~U~U")
(define *filter-external-comment* ignore)
(define *filter-external-body* identity)
```

3 Extending the pretty-printer

The pretty-printer is perfectly naive but fulfills many needs. By default, it is implemented approximatively as the `write` Scheme function. A bunch of defining forms exist to extend the pretty-printer. These forms may be evaluated through the `\Eval` directive or gathered in a file which is preloaded if it appears as an option preceding the `--` option. These files generally have the `.l2t` extension. Some of these files belong to the official distribution of L2T and mainly `default.l2t` which is used for this documentation.

For all the following defining forms, we will use a running example. Suppose we want to pretty-print the following definition:

```
(define (extend* f x* y*)
  (if (pair? x*)
      (if (pair? y*)
```

```
        (extend (extend* f (cdr x*) (cdr y*)) (car x*) (car y*))
        (wrong "Not enough images") )
    (if (null? y*)
        f
        (wrong "Too much images") ) ) )
```

By default, the pretty-printer uglily displays the previous definition. As you may see, if you do nothing for the pretty-printer, it will not do anything worth for you!

$(define(extend*fx*y*)(if(pair?x*)(if(pair?y*)(extend(extend*f(cdrx*)(cdry*))(carx*)(cary*))(wrongNotenoughima$

(def-type-printer *predicate method*)

This directive specifies the method to use for printing the objects that satisfies the *predicate*. A number of predefined methods exist in the pretty-printing engine that can be used for that purpose: for instance, the `pp-default` method corresponds to the primitive behavior of the pretty-printer. Method generators also exist and direct native methods can also be written.

Examples

A method exists that pretty-prints pairs i.e. forms as mathematical function calls i.e. `(f x y)` appears as $f(x, y)$. This method is named `pp-pair`. One can use it after specifying:

```
(def-type-printer pair?    pp-pair)
```

Our running example then becomes:

$define(extend*(f, x*, y*), if(pair?(x*), if(pair?(y*), extend(extend*(f, cdr(x*), cdr(y*)), car(x*), car(y*)), wrong(Noten$

Predefined methods

Among predefined methods are:

`pp-pair` prints forms like `(f x y)` as $f(x, y)$.

`pp-list` prints forms like `(f (g x) y)` as $(f\ (g\ x)\ y)$. It does not use commas therefore it mimics better λ-calculus.

`pp-boolean` prints booleans in boldface and full names as in **false** and **true**.

`pp-symbol` disambiguates then prints symbols, see below the `def-image` directive.

`pp-char` prints characters surrounded with quotes as in ‘z’

(def-form-printer *keyword method*)

This directive declares how to print a form with *keyword* as its first term. It is mainly used for special forms or functions that need to be particularly displayed. A number of predefined methods exist in the pretty-printing engine that can be used for that purpose. Method generators also exist and direct native methods can also be written.

Method generators are functions that returns appropriate methods.

`(pp-nullary` *`image`*`)` is used to generate a method that will take a form without arguments. The behavior of this method is to simply print the *image*.

`(pp-unary` *`prefix suffix`*`)` is used to generate a method that will take a form with two terms i.e. a unary function and its argument. The behavior of the method is to print the argument surrounded by *prefix* and *suffix*. The *prefix* and *suffix* are regular strings.

`(pp-binary` *`beginning middle end`*`)` is used to generate a method that will take a form with three terms i.e. a binary function and its two arguments. The behavior of the method is to print the *beginning* prefix, then the first argument, then the *middle* string, then the second argument finally followed by the *end* string.

`(pp-ternary` *`beginning middle1 middle2 end`*`)` Similar to the previous ones except that it takes care of ternary functions.

`(pp-nary` *`beginning separator end`*`)` is used to generate a method that will take forms with any number of terms. The behavior of the method is to print the *beginning* string, then each argument followed by the *end* string. Between any two arguments is printed the *separator* string.

`(pp*-nary` *`strings ...`*`)` is used to generate a method that will take a form with a one term less than the number of *strings*. The behavior of the method is to print the first *string* then the first term, then the second *string* followed by the second term etc. The last term of the form is followed by the last *string*. An error is raised if the number of terms is incorrect with respect to the number of *strings*.

Observe for all these method generators that the term in functional position is not printed at all.

The pretty-printing engine also defines numerous methods for usual special forms such as `if`, `let`, `define` which name is regularly made with the name of the special form prefixed with `pp-`. Other utilities also exist such as `pp-print-terms` or `pp-print-bindings`, see the `pp.scm` file for more details.

Examples

The `default.l2t` contains a lot of predefined uses of these method generators. Here are some of them:

```
(def-form-printer =       (pp-binary "" " = " ""))
(def-form-printer car     (pp-unary "" "\\!\\downarrow_1"))
(def-form-printer list    (pp-nary "<" "," ">"))
(def-form-printer cons    (pp-binary "<" ">\\S" ""))
(def-form-printer extend
  (pp-ternary "" "\\lbrack " "\\rightarrow " "\\rbrack ") )
(def-form-printer extend*
  (pp-ternary "" "\\lbrack "
             "\\stackrel{\\ast}{\\rightarrow} "
             "\\rbrack " ) )
```

With all the definitions of `default.l2t`, our running example looks better:

$f[x* \stackrel{*}{\rightarrow} y*] =$
$\quad \textbf{if}\, x* \neq <>$
$\quad \textbf{then}\ \ \textbf{if}\, y* \neq <>$
$\qquad\qquad \textbf{then}\, f[x*\dagger 1 \stackrel{*}{\rightarrow} y*\dagger 1][x*\downarrow_1 \rightarrow y*\downarrow_1]$
$\qquad\qquad \textbf{else}\, \mathit{wrong}($“`"Not enough images"`”$)$

endif
else **if** $y* =<>$
then f
else $wrong$(“"Too much images"”)
endif
endif

(def-image "*string*" *variables ...*)

This directive declares how must be printed the specified variables: they are printed as specified by the string. Since many different variables may be specified a common image, some disambiguating schemes are predefined.

Customization

The variable `*preferred-index*` rules how to disambiguate variables that have a common image. Four values are possible:

`nothing` This mode does not disambiguate variables but it warns the user if a clash is encountered i.e. if two different variables are bound in the same binding form and have a similar image.

`index` This mode disambiguates variables with subscripts. Variables are printed as v_0, v_1, v_2 etc.

`then-index` This mode disambiguates variables with subscripts except the first one which is printed without subscript. The first variables are thus printed as v, v_1, v_2 etc. This is the default mode.

`prime` This mode disambiguates variables with suffixed quotes. The first variables are thus printed as v, v', v'' etc.

The `*preferred-index*` imposes the default indexing mode but this can be specified on a per variable basis using `def-indexing-mode.`

(def-indexing-mode *mode variables ...*)

This directive declares how must be disambiguated the specified variables. The mode must be one of the four previously described modes.

Example

Back to our running example, we can specify that the variable `f` should be displayed by a greek φ while `x*`, `y*` will be converted, for the sake of the experiment, into a common image: ε^*.

The following result is specified as (with `x*` and `y*` disambiguated with primes instead of subscripts):

```
(def-image "\\varphi " f)
(def-image "\\varepsilon^\\ast " x* y*)
```

```
(def-indexing-mode prime x* y*)
(def-type-printer symbol?  pp-symbol)
```

And the result now appears as:

$\varphi[\varepsilon^* \xrightarrow{*} \varepsilon^{*\prime}] =$
$\quad$**if** $\varepsilon^* \neq <>$
$\quad$**then** **if** $\varepsilon^{*\prime} \neq <>$
$\qquad\quad$**then** $\varphi[\varepsilon^* \dagger 1 \xrightarrow{*} \varepsilon^{*\prime} \dagger 1][\varepsilon^* \downarrow_1 \rightarrow \varepsilon^{*\prime} \downarrow_1]$
$\qquad\quad$**else** *wrong*(“`"Not enough images"`”)
$\qquad\quad$**endif**
$\quad$**else** **if** $\varepsilon^{*\prime} = <>$
$\qquad\quad$**then** φ
$\qquad\quad$**else** *wrong*(“`"Too much images"`”)
$\qquad\quad$**endif**
$\quad$**endif**

Controlling line breaks

The pretty-printer breaks lines too often but is expected to do so since it is easier to remove these line-breaks rather than to insert them. To save space, let us take the definition of `extend` which yields:

```
(define (extend f x y)
  (lambda (u)
    (if (equal? u x) y (f u)) ) )
```

The generated code of the previous example is:

```
%%%
\DenotedLisp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Definition of extend
{\varphi }\lbrack {\mbox{\it x\/}}\rightarrow {\mbox{\it y\/}}\rbrack  = \newline
{\mbox{\it lambda\/}}({\mbox{\it u\/}}(), \setandincrindent
\:\mbox{\bf{if}}\:{\mbox{\it u\/}} = {\mbox{\it x\/}} \newline
\:\mbox{\bf{then}}\:{\mbox{\it y\/}} \newline
\:\mbox{\bf{else}}\:{\varphi }({\mbox{\it u\/}}) \newline
\:\mbox{\bf{endif}}\: \decrindent
) %%%
\EndDenotedLisp
```

It is therefore easy to comment the `\newline` that appear at the end of lines. For this simple case, it is even easier to use the `\DenotationInlinetrue` TEX macro which is tested in the definition of `\DenotedLisp` TEX macro and directly obtain:

$\varphi[x \rightarrow y] = \lambda u.$ **if** $u = x$ **then** y **else** $\varphi(u)$ **endif**

Back to denotations

Denotational style also use valuation i.e. functions converting syntax into λ-terms. There again L2T offers some simple support. Two method generators exist to help. They both take as first argument a curly letter(s) to name the valuation. This argument is followed by strings that will recreate the syntactical form of the denoted program. The so-called semantical brackets are defined by the variables `pp-semantical-lbracket` an `pp-semantical-rbracket`.

(pp-meaning *curly-letter prefix separator end*) prints the *curly-letter* followed by the opening semantics $[\![$ bracket followed by the prefix. Each term of the form is then displayed separated by *separator* and finally followed by *end* itself followed by the closing semantics bracket $]\!]$. This method is reminiscent of `pp-nary`.

(pp*-meaning *curly-letter prefix separator1 separator2 ... end*) prints the *curly-letter* followed by the opening semantics bracket $[\![$ followed by the prefix. Each term of the form is then displayed and followed by the associated separator. All the separators must be used. Finally the suffix *end* is emitted followed by the closing semantics bracket $]\!]$. This method is reminiscent of `pp*-nary`.

Here is a simple example. We define an extension to Scheme where `(dynamic` *name*`)` yields the dynamic value associated to *name*; a value is associated to a name with `(dynamic-let (`*name value*`)` *body*`)`. The denotations are defined by the following regular Scheme code:

```
(define (meaning-dynamic-reference n)
   (lambda (r d k s) (k (d n) s)) )
(define (meaning-dynamic-binding n e e+)
  (lambda (r d k s)
    ((meaning e) r d
                (lambda (v s1)
                  ((meaning*-sequence e+)
                   r (extend d n v) k s1 ) )
                s ) ) )
```

The associated syntaxes are defined with:

```
(def-form-printer meaning
  (pp-meaning "\\cal E" "" "" "") )
(def-form-printer meaning*-sequence
  (pp-meaning "\\cal E^\\ast" "" "" "") )
(def-form-printer meaning-dynamic-reference
  (pp-meaning "\\cal E" "\\mbox{\\tt(global }" "\\mbox{\\ }" "\\mbox{\\tt)}") )
(def-form-printer meaning-dynamic-binding
  (pp*-meaning "\\cal E" "\\mbox{\\tt(dynamic-let (}" "\\mbox{\\ }"
                         "\\mbox{\\tt) }" "\\mbox{\\tt)}" ) )
(def-image "\\nu" n)
(def-image "\\rho" r)
(def-image "\\delta" d)
(def-image "\\kappa" k)
(def-image "\\sigma" s s1)
(def-image "\\pi" e)
(def-image "\\pi^{+}" e+)
```

So we get the fine result:

$$\mathcal{E}[\![\texttt{(global }\nu\texttt{)}]\!] = \lambda\rho\delta\kappa\sigma.\kappa(\delta(\nu),\sigma)$$

$$\mathcal{E}[\![\texttt{(dynamic-let (}\nu\ \pi\texttt{) }\pi^{+}\texttt{)}]\!] = \lambda\rho\delta\kappa\sigma.\mathcal{E}[\![\pi]\!](\rho,\delta,\lambda\varepsilon\sigma_\infty.\mathcal{E}^*[\![\pi^+]\!](\rho,\delta[\nu\rightarrow\varepsilon],\kappa,\sigma_\infty),\sigma)$$

Hand-written methods

Methods can be also directly written. A method is a ternary function which receives the expression to print, the current environment (in which are stored methods, images of variables, indentation level etc.), and the output port. Read the source file `pp.scm` to write such methods and use auxiliary functions such as `pp-print`, `pp-print-terms` and `pp-format`.

Format directives

The `pp-format` function is similar to the usual `format` function but only recognizes the following format directives:

`~A` or `~a` to display an expression,

`~S` or `~S` to write an expression,

`~U` to display a string without enclosing quotes,

`~Q` or `~q` to greeky-print an expression,

`~G` or `~g` to pretty-print an expression (this uses Marc Feeley's pretty-printer),

`~~` to print a tilde,

`~%` to insert a newline.

`~*` to swallow the argument without showing it.

Marc Feeley's pretty-printer engine has been modified to be parameterized over the string that is inserted whenever a line is broken. This is the `*inter-line-string*` variable, I mainly use it in conjunction with the `\ShowEvaluationOf` directive. Pay attention, the value of this variable must be a raw string, it is not interpreted as a format string.

Another variable `*pretty-line-width*` is the width of the line where to pretty-print.

```
(set! *inter-line-string* (make-string 1 #\newline))

(define *pretty-line-width* 60)
```

To program methods allows to obtain various effects. For instance, the `number.l2t` file allows to subscript lists by their depth. It uses the `pp-context` function that returns the list of all the forms, the current form is contained in (here we subscript each list with the length of its context). The function `pp-context` can be used to improve line breaking taking care of the context. Here is a small example of the `number.l2t` customization file.

$\texttt{(a(b((c)}_3\texttt{)}_2\texttt{ (((d)}_4\texttt{)}_3\texttt{)}_2\texttt{)}_1\texttt{ e)}$

Not only functions but macros can also be defined within customization files. To be portable across various compilation of L2T, these macros must be defined with `define-lisp2tex-macro`. There is an example in the `domain.l2t` customization file. If we use this macro, we can simply say:

`(def-sum-domain (Value Value) (Function Function) (Integer Integer))`

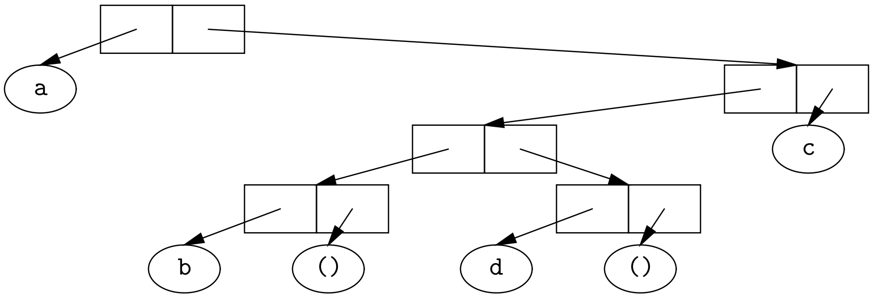
and try it on the expression:

`(Value->Function (inValue 42))`.

The result is:

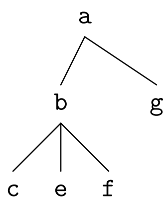
$\mathit{in}\mathbf{Value}(42)\ |_{\mathbf{Function}}$

Here is another customization example, it uses `gpic` the Gnu version of Pic from Brian Kernighan [KR91]. The `sexp.l2t` customization file allows to generate graphical images of S-expressions. The generated TEX file must be filtered by `gpic -t`. The generated TEX is parameterized with `gpic` macros. Suppose you want to pretty-print `(a ((b) d) .  c)` then you get the following figure, simply writing `\PrettyPrint (a ((b) d) .  c)`.



Other customization files are provided in the distribution of L2T:

- `default.l2t`, this is the customization file used to process this documentation. It is given as example.
- `pp.l2t`, this is a customization file that uses the real pretty-printer of the underlying implementation to pretty-print expressions. It is also given as an example.
- `qnc.l2t`, this is my once favorite customization file, it takes into account my favorite LaTeX style file (see the description of `ApocaLIXP.sty` below).
- `domain.l2t`, this an example of a macro in a customization file.
- `tree.l2t`, it displays (with `gpic`) S-expressions as trees. The S-expression `(a (b c (e) f) g)` will be displayed as:



4 The snippet style

By default, L2T wraps its output within TeX macros. Inline snippets are wrapped within `\snip` while block excerpts (file excerpts) are wrapped within `\snippet`. Scheme pretty-printed forms are wrapped within special macros. Citations are made via `\snippetCite`. The snippet style is now the default style used by L2T. The snippet style exists for LaTeX and for Hevea to produce HTML code.

4.1 The `snippet` command

Code excerpts start with the `\snippet` command and end with `\endsnippet`. The `\snippet` command takes an optional argument (embedded within square brackets) that specifies the default language. Among these languages are `c` and `scheme` (but you may define your own possibly inheriting from these). The language is used to provide a default customization.

Within the excerpt, all characters (including newlines) are copied verbatim. There is however a single exception: commands starting with a backslash are recognized and processed but these commands cannot have arguments. Apart the `\endsnippet` nullary command that ends the `\snippet` quasi-environment, there is also the `\[` ... `\]` sequence that typesets its inner text in LaTeX.

Observe that these commands do not form a pure LaTeX environment so you cannot write `\begin{snippet}` ...`\end{snippet}`. This is due to TeX and its `catcode` behavior. It is simpler to recognize the end with a regular command than something containing curly braces. However it is simple to transform one into the other if needed with help of some `sed` or `perl` script.

As a matter of style, it is better to start the code to be copied verbatim on the line that follows the `snippet` command. This is to avoid a first line which is not aligned with the following ones.

The default language (which is, by default, the current value of the `snippetDefaultLanguage` command i.e. the `scheme` language) is obtained when one does not mention any language; in this case you even don't need to use the square brackets since there is nothing in it. You thus may write directly:

```
This    a
     is   snippet


This    another
     is   snippet


This    a third
     is   snippet
```

The source code for the previous example shows the various possible wordings. It is however encouraged that you always use an explicit language to qualify the snippet because this language will be used to give a class for the DIV section in the generated HTML. Here is the source code:

```
\snippet
This    \[a\]
     is   snippet \endsnippet \snippet[]
This    \[another\]
     is   snippet \endsnippet \snippet[scheme]
This    \[a third\]
     is   snippet \endsnippet
```

Of course it is possible to define one's own style (see below). Even if it is possible not to mention the language, it is far better to always specify it to obtain a better rendering into HTML. This is because the language is used to give a CLASS tag to the DIV tag that encloses a snippet. You may then associate to this CLASS some rendering properties via a Cascading Style Sheet file.

Customization

There is a number of ways to customize the rendering of snippets. Customization is expressed by hooks. There are general hooks and language-dependent hooks. These later hooks are run after the general ones so they may correct the default behavior for the hook. Therefore there are two different levels of customization depending on whether you want to customize the snippet engine or a particular language rendering.

snippetBeforeHook is run when entering the snippet engine just after the `\snippet` command (and its optional argument) but before any character is read. By default, it starts a new paragraph.

snippetAfterHook is run when exiting the snippet engine just before the `\endsnippet` command. By default, it closes the current paragraph.

snippetBeforeLineHook is run at the beginning of every line. It may be redefined to print the number of the line for instance. By default, it does nothing.

snippetAfterLineHook is run at the end of every line. By default it does nothing.

snippetBeforeTeXHook is run before any TeX inclusion opened by the `\[` command. By default it changes the font to italics.

snippetAfterTeXHook is run after any TeX inclusion closed by the `\]` command. By default, it closes the math mode if not yet done.

snippetRestartHook is an internal hook which is run any time the snippet engine is restarted.

snippetVerbatimHook is an internal hook which is run any time the catcode of characters are changed so it is possible to make some characters active. The Scheme environment for instance redefines the semicolon to render comments in italics.

For all these hooks, there may exist a language dependent hook whose name is formed after the name of the hook suffixed by the name of the language (the optional argument of the `\snippet` command which is recorded in the `\snippetLanguage` command which, by default, is equal to the current value of the `snippetDefaultLanguage`) command.

Moreover, styles may be inherited. The exact behavior of the hook mechanism is as follows. Suppose the optional language of the snippet be *abc* and the hook *hook* then, *hook* is run (if defined) then *hooka*, *hookab* and *hookabc*. Therefore if you mention `schemeplus` as the language of a snippet, you will run the general hooks, the `scheme` hooks and the `schemeplus` hooks (and all sort of intermediate hooks such as `sch` or `schemeplus` if they are defined).

It is often the case that within these hooks you only toggle some predefined customization habits. Here there are:

snippetEnableTex defines the macros delimiting an insertion of plain TeX within a snippet. It is custom to use `\[` and `\]` to enter and exit the inclusion.

enableLispComment renders Lisp comments in TeX.

disableLispComment renders Lisp comments verbatim.

keepLeadingSpaces (a condition) keeps leading spaces right. after the semi-colons beginning a Lisp comment.

enableBackSlashDoubleQuote makes `\"` appear as `\"`.

enableSchemeCharacters makes `#\c` appears as `#\c`.

enableDoubleBackSlash makes `\\` appears as they stand. This is useful within Scheme strings.

disableSchemeCharacters makes active a macro after a hash sign.

enableSharpInComment makes # to be a regular character within comments. This makes easy to talk of true and false!

enableSomeBackslashedChars (in C) makes `\n` and siblings appear as they stand.

showBackslashNewline (in C) makes a backslash followed by a newline appears as a backslash alone at the end of a line.

doubleBackslashToLetOneAppear (in Perl) makes `\\` stands as it is.

Here is a small example of the snippet style. First the source code:

```
\snippet[scheme]
;;;\relax This is the \texttt{code} of:
(de \TeX (foo) \[\hfill \LARGE\lispfont [Function]\]
   (bar 'hux) ; comment $for$ Lisp
   \[\it \fbox{$2^{31}$} \] ) \endsnippet
```

This yields:

```
;;; This is the code of:
(de TeX (foo)                                   [Function]
   (bar 'hux) ; comment for Lisp
   [2^31] )
```

4.2 The `snip` command

The `\snip` command permits the insertion of code within lines. It is somewhat similar to the `\verb` command except that it shares a number of features of the `\snippet` command.

A typical use of the `\snip` command is the following: `\snip[file]{file.html}` where the word *file.html* will be typeset according to the convention of the *file* style. By default, if the optional language parameter is absent, it defaults to the value of the `snippetDefaultLanguage` command. Therefore to write `\snip[\snippetDefaultLanguage]{file.html}` is equivalent to write more simply `\snip{file.html}`.

The following hooks are defined for the `\snip` command:

snipBeforeHook which is run right before,

snipAfterHook which is run right after.

5 SLaTeX compatibility

Other styles can probably be written to generate $\mathcal{STOL}$ or SLaTeX text.

SLaTeX may now be handled by L2T. If you want to use Dorai Sitaram's pretty-printer, load the `slatex.l2t` customization file and just use the regular `\Print` and `\FromFile` directives of L2T. Don't forget to run SLaTeX on the output! You may revert from and back to SLaTeX style with `\Eval`(slatex 'off) or `\Eval`(slatex 'on). These previous S-expressions were printed by SLaTeX as are the following parameters controlling SLaTeX generation in SLaTeX style:

```
(define *beginslatex-block* "\\begin{schemedisplay}~%")
(define *interslatex-block* " ~%")
(define *endslatex-block*   " \\end{schemedisplay}")
(define *slatex-excerpt* "{\\scheme;~A;}")
```

6 Miscellaneous

To ease the use of `make`, it is better to select an extension for the files that must be filtered by L2T. Since the name is long and difficult to type[4], I had to choose a simpler extension. For historical reasons I selected

[4] Actually I use `tcsh` and it is the only command that starts with `Li`, it is therefore a simple matter to make `tcsh` complete this name.

bk for "book". The rule to regenerate a TEX file therefore is, where LIBDIR is the path to the library of l2t files (usually /usr/local/lib/LiSP2TeX/:

```
LiSP2TeXFLAGS = -v -I${LIBDIR}
.SUFFIXES: .tex .bk
.bk.tex : ; LiSP2TeX ${LiSP2TeXFLAGS} -- $*.bk > $*.tex
```

The usual extension for customization files is l2t. Since it is not checked, you can use your own ones.

6.1 Dirty tricks

The \endinput stops TEX to process a file. You may achieve a similar effect in L2T by writing something like:

```
\Eval (*exit*)
```

7 Contributions

Included in the distribution is a directory named *contrib* that contains additional code making use of l2t.

Such a code is sxmlp by Kirill Lisovsky who binds Oleg Kiselyov's SSAX XML parser and l2t to easily process XML documents. This code depends on Gambit as underlying Scheme system (that is precisely the Scheme system used in the *rpm* distribution).

8 Conclusion

This program is very simple but powerful and relatively fast. I hope that the new ideas of separating the files but combining them at will will satisfy those who have to document files that they do not want to alter.

References

[IEE91] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[KH92] Daniel Kobler and Daniel Hernández. $\mathcal{STOL}$ – literate programming in Scheme. *Lisp Pointers*, 5(4):21–30, 1992.

[KR91] Brian W Kernighan and Dennis Ritchie. Pic – a graphics language for typesetting. Computing Science Technical Report 116, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, May 1991.

[Que91] Christian Queinnec. Meroon: A small, efficient and enhanced object system. Technical Report LIX.RR.92.14, École Polytechnique, Palaiseau Cedex, France, November 1991.

[Ste90] Guy L. Steele, Jr. *Common Lisp, the Language*. Digital Press, Burlington MA (USA), 2nd edition, 1990.

[Wei90] Pierre Weis. The CAML reference manual, version 2.6.1. Technical Report 121, Institut National de Recherche en Informatique et en Automatique, 1990.