1 Projects

(NO)WEB HYPERTEXT SYSTEM IN EMACS (WHYSE) is a project-based application. Projects are lists registered with WHYSE using the "Easy Customization Interface", which provides a simple way to make the necessary information known to WHYSE. Users register a literate programming project (only Noweb-based programming is supported) as an item in the customization variable `w-registered-projects`; further project data is contained in a Common Lisp struct during runtime.

In short, a project is composed of several things:

- a name,
- a Noweb source file,
- a shell command to run a user-defined script
- an SQLITE3 database, and a connection thereto,
- a frame,
- and date-time information (creation, edition, and export).

The struct keeps some information during runtime, like the connection, but other information is generated at runtime (such as the filename of the database). These items are each explained in this section. If some item is not well-enough explained in this section, please try editing the Noweb source and improving the explanation and creating a pull-request against the WHYSE Emacs Lisp repository on its Git forge; you may also submit your edition by email to the package maintainer.

Users of WHYSE in Emacs are expected to be familiar with Noweb; this does not include how Noweb is built from source (that is arcane, supposedly) or how filters are implemented with Sed, AWK, or other languages. Users must know, however, how to write a custom command-line for noweave (read the manual section regarding the `-v` option).

Developers of WHYSE extensions (in either SQL or Emacs Lisp) should read the Noweb Hacker's Guide until they understand it, afterwards reading this documentation several times until the full implementation is understood. I recommend modifying the system using itself to keep organized, and writing literately; you'll thank yourself later for doing so.

A customization group for WHYSE is defined to organize its customization variables, and these details are explained before moving on to explain the struct used during runtime.

1 ⟨*Customization and global variables* 1⟩≡ (23a) 4c▷

```
(defgroup whyse nil
  "noWeb HYpertext System in Emacs"
  :tag "WHYSE"
  :group 'applications)

(defcustom w-registered-projects nil
  "This variable stores all of the projects that are known to WHYSE."
  :group 'whyse
  :type '(repeat w-project-widget)
  :require 'widget
  :tag "WHYSE Registered Projects")
```

```
(defvar w-parse-success t
  "A simple boolean regarding the success or fialure of the last attempt to parse a buffer of Noweb tool
```

Defines:
w-parse-success, used in chunk 12a.
w-registered-projects, used in chunk 4.
whyse, used in chunks 4a and 21.

The Widget feature is required by the registered projects variable, but may be redundant because the Easy Customization Interface is itself implemented with The Emacs Widget Library. Requiring the library may be undesirable, as `(require 'widget)` will be eagerly evaluated upon Emacs' initialization when `w-registered-projects` is set to its saved custom value. However, there may be a good reason to eagerly evaluate that form: the Widget feature will be available immediately, and widgets will be used in buffers to provide TUI buttons for navigation between modules of a literate program (at least, that is the design of the program at this point in development), so having this feature available sooner than later is okay. The feature is required by the package regardless.

The `w-project-widget` type used for the registered projects variable is a simple list widget containing the name of the project and its Noweb source file, along with a filename for a shell script which generates the Noweb tool syntax for this project. Each Noweb project has a different command-line, and some are complex enough to have a makefile, or multiple makefiles! Noweb itself is an example of that level of complexity. The shell script is later executed by WHYSE upon loading the project, and the standard output captured for parsing by a PEG parser.

3 ⟨*Widgets* 3⟩≡ (23a)

```
(define-widget 'w-project-widget 'list
  "The WHYSE project widget type."
  :format "\n%v\n"
  :offset 0
  :indent 0

  ;; NOTE: the convert-widget keyword with the argument
  ;; 'widget-types-convert-widget is absolutely necessary for ARGS to be
  ;; converted to widgets.
  :convert-widget 'widget-types-convert-widget
  :args '((editable-field
           :format "%t: %v"
           :tag "Name"
           :value "")

          (file
           :tag "Noweb source file (*.nw)"
           :format "%t: %v"
           :valid-regexp ".*\\.nw$"
           :value "")

          (string
           :tag "A shell command to run a shell script to generates Noweb tool syntax"
           :format "%t: %v"
           :documentation "A shell script which will produce the
           Noweb tool syntax. Any shell commands involved with
           noweave should be included, but totex should of course
           be excluded from this script. The script should output
           the full syntax to standard output. See the Noweb
           implementation of WHYSE for explanation."
           :value "")))
```

NB: Comments may be superfluous in a literate document like this, but some effort was made to produce a readable source file regardless of the general principles of literate programming; other authors write warnings into their tangled source files: “Don’t read this file! Read the Noweb source only!”. I don’t say that, especially for an Emacs application.

The sole interactive command—`whyse`—loads the first element of `w-registered-projects`, considering it the default project.

4a ⟨*Quotation custom-set-variables* 4a⟩≡
```
'(w-registered-projects
  '(("Noweb Hypertext System"
     "~/Desktop/whyse.nw"
     "make -C ~/Desktop -silent -file ~/src/whyse/Makefile tool-syntax"))
  nil
  (widget))
```
Uses `w-registered-projects` 1 and `whyse` 1 4b.

The `whyse` command is very simple, it checks if projects have been defined in the Customize interface, and if so uses the first one to *initialize a new project*. If no projects are defined or there are any nil values in the procedure, the Customize interface is opened.

4b ⟨*WHYSE* 4b⟩≡ (23a)
```
(defun whyse ()
  (interactive)
  (if-let ((w-load-default-project?)
           (default-project (cl-first w-registered-projects))
           (project (make-w-project :name (cl-first default-project)
                                    :noweb (cl-second default-project)
                                    :script (cl-third default-project))))
      ⟨convert the Noweb to tool format and parse it with the PEG 6a⟩
      ⟨open Customize to register projects 5a⟩))
```

Defines:
`whyse`, used in chunks 4a and 21.
Uses `w-load-default-project?` 4c and `w-registered-projects` 1.

WHYSE is likely to be useful for very large literate programs, so the command is designed to initialize from an existing project without prompt. In more verbose terms: unless `w-load-default-project?` is non-nil and `w-registered-projects` includes at least one element, Customize will be opened to customize the WHYSE group when `whyse` is invoked.

4c ⟨*Customization and global variables* 1⟩+≡ (23a) ◁1
```
(defcustom w-load-default-project? t
  "Non-nil values mean the system will load the default project.

nil will cause the interactive command ‘whyse’ to open Customize on
its group of variables."
  :type 'boolean
  :group 'whyse
  :tag "Load default project when ‘whyse’ is invoked?")
```

Defines:
`w-load-default-project?`, used in chunks 4b and 5a.

5a ⟨*open Customize to register projects* 5a⟩≡ (4b)

```
(message
 "No WHYSE projects registered, or ‘w-load-default-project?’ is nil. %s"
 (customize-group 'whyse))
```

Uses `w-load-default-project?` 4c.

5b ⟨*WHYSE project structure* 5b⟩≡ (23a)

```
(cl-defstruct w-project
  "A WHYSE project"
  ;; Fundamental
  name
  noweb
  script
  database-file
  database-connection

  ;; Usage
  frame

  ;; Metadata
  (date-created (ts-now))
  date-last-edited
  date-last-exported

  ;; TODO: limit with a customization variable so that it does not grow too large.
  history-sql-commands)
```

Instances of this struct are only initialized with a few values: `name`, `noweb`, and `script`. The rest of the fields either have default values dependent upon the input data (like the database-file, database-connection, and date-created), or are given values when appropriate later in operation (such as `date-last-exported`) or upon initialization (`frame`).

Initialization when the interactive command is called is covered next; to summarize: `w-project-load-hook` is run.

2 System initialization from new projects

To summarize this section, since it is longer than the previous section, the object is the definition of `«convert the Noweb to tool format and parse it with the PEG»`, which is a chunk used in `whyse`.

In more explicit words, this section describes the actions that occur when a user invokes `whyse` interactively (with *M-x*) and the preconditions have been met; the `whyse` function has already been introduced, and only the "meaty" business end of its operation has been left undefined until now. Ergo, `«convert the Noweb to tool format and parse it with the PEG»` gathers together the functionality that converts a Noweb to its tool syntax with a project's specified shell script, and parses the text before the next section of body forms is executed. Those send the parsed text to the database, and finally create the atomic window for the IDE in the active frame.

6a ⟨*convert the Noweb to tool format and parse it with the PEG* 6a⟩≡ (4b)

```
(with-temp-buffer
  (insert (shell-command-to-string (w-project-script project)))
  (goto-char (point-min))
  (w-parse-current-buffer-with-rules))
```

Uses `w-parse-current-buffer-with-rules` 12a.

2.1 Conversion to tool syntax

WHYSE could have been written to call the `noweave` programs itself, but that is less configurable than providing the opportunity to let the user configure this on their own. It respects Noweb's pipelined architecture, and keeps things as transparent as possible. What should be written in Emacs Lisp is written therein, and what shouldn't be implemented in Elisp is not. The tool syntax is thus obtained by running the shell script configured for the project by calling it with the command-line provided in the third element of an entry in `w-registered-projects`.

6b ⟨*run the project shell script to obtain the tool syntax* 6b⟩≡

```
(make-process
 :name "w-tool-generation"
 :buffer (get-buffer buffer)
 :command `("bash" ;; likely BASH on a GNU system, hoping for the `command-string' option.
            "-c"
            (,@(w-project-script project)))

 :stderr (generate-new-buffer "WHYSE tool generation standard error stream")
 :sentinel (lambda (process event-string)
             (message "%S: %s" process event-string)))
```

The PEG for Noweb's tool syntax is run on the result of the shell script, and this value consumed by the parent of this chunk.

2.2 Database initialization

Every project should have a database file located somewhere within the user's Emacs directory; if the user is a Spacemacs user, then Spacemacs' cache directory is used, otherwise the database is made in the user's Emacs directory and not a sub-directory thereof.

The form used to create the absolute path for the location of the database joins three things: the user's Emacs directory, `nil` or Spacemacs' cache directory, and the name of the project with ".db" appended. Note that concatenating `nil` with a string is the same as returning the string unchanged.

7a ⟨*return a filename for the project database* 7a⟩≡ (7b)

```
(file-name-concat
 ;; Usually ~/.emacs.d/
 user-emacs-directory
 ;; 'nil' or the Spacemacs cache directory.
 (when (f-directory? (expand-file-name ".cache" user-emacs-directory))
   ".cache")
 ;; PROJECT-NAME.db
 (concat (w-project-name project)
         ".db"))
```

For SQLITE, the path name of the database to connect to or create is sufficient to establish a connection, so the next step is to connect to the database and store the connection object in the appropriate slot of the project struct.

7b ⟨*create the database* 7b⟩≡

```
(setf (w-project-database-connection project)
      (emacsql-sqlite
       (w-project-database-file ⟨return a filename for the project database 7a⟩))))
```

The only thing left to do is establish the schema of the tables, which is done by mapping over several EMACSQL s-expressions.

8 ⟨*map over SQL s-expressions, creating the tables* 8⟩≡

```
(-map (emacsql (w-project-database-connection project) it)

      ;; A list of SQL s-expressions to create the tables.
      '([:create-table module
         ([module-name
           content
           file-name
           section-name
           (displacement integer)
           (module-number integer :primary-key)])]

        [:create-table parent-child
         ([(parent integer)
           (child  integer)
           (line-number integer)]
          (:primary-key [parent
                         child]))]

        [:create-table identifier-used-in-module
         ([identifier-name
           (module-number integer)
           (line-number integer)
           type-of-usage]
          (:primary-key [identifier-name
                         module-number
                         line-number
                         type-of-usage]))]

        [:create-table topic-referenced-in-module
         ([(topic-name nil)
           (module-number integer)]
          (:primary-key [topic-name
                         module-number]))]))
```

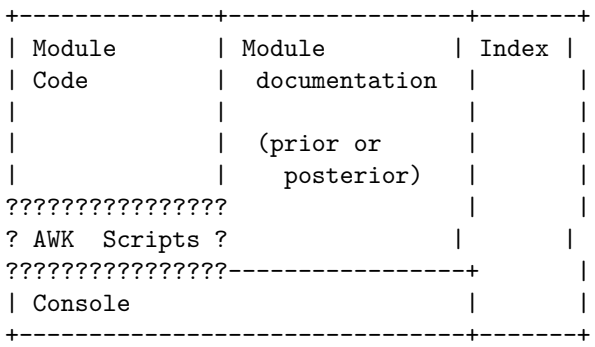

Figure 1: Simple drawing of WHYSE frame layout

2.3 Frame creation and atomic window specification

A frame like in Figure 2.3 should be created.

9 ⟨*Get project frame* 9⟩≡

```
(progn
  (select-frame (w-project-frame project))
  (switch-to-buffer (generate-new-buffer (w-project-name project))
                    nil
                    'force-same-window)
  (let* ((window-right (split-window-right))
         (parent-window (window-parent window-right)))
    (window-make-atom parent-window)
    (display-buffer-in-atom-window
     (get-buffer-create (format "Module Index<%s>"
                                (w-project-name project)))
     `((window . ,parent-window) (window-height . 8)))))
```

3 System initialization from existing projects

WHYSE loads a project by running the shell script stored in the third element of the project list (which is pointed to by the script slot in the struct).

3.1 Initializing from an existing project

With a default project available, WHYSE runs `w-project-load-hook` with the struct of the default project let-bound as `project`. Much of the functionality of WHYSE is implemented with the default hook, and extensions to WHYSE should be implemented by editing the WHYSE Noweb source and recompiling it, or extending the existing system with more hook functions added to the aforementioned hook list variable.

If the project's database file is empty (zero-bytes) or does not exist then the database is created from scratch. If the database already exists, the first module is loaded and the database is not changed.

10 ⟨*delete the database if it already exists, but only if it's an empty file* 10⟩≡

```
;; Unless the SQLite database's size is zero or it doesn't exist, move it to the user's trash directory.
(let ((w-dbfile (w-project-database-file project)))
  (unless (or (not (file-exists-p w-dbfile))
              (= 0 (file-attribute-size (file-attributes w-dbfile))))

    ;; TODO: Is there a better way to do this? 'backup-buffer'?
    (copy-file w-dbfile (concat w-dbfile "~") t)

    ;; TODO: ensure that this AREA of code is reasonable before release.
    ;; It may have been written to ease development only.
    (let ((delete-by-moving-to-trash t))
      (delete-file w-dbfile t)))
```

4 Loading Noweb source files

To parse a noweb source file, the file needs to be loaded into a temporary buffer, then it can be parsed.

A simple usage of NOWEB is given next, which shows that `noweave` does not include the header keyword, nor autodefinitions, usages, or indexing by default. Those are further stages in the UNIX pipeline defined by the user with `noweave` command-line program options and flags.

The WHYSE system parses the tool syntax emitted by `markup`, and early development versions (prior to version `0.n-devel`) completely ignore Noweb keywords out of that scope.

An example of a NOWEB command-line a user may call is given next.

```
[bryce@fedora whyse]$ noweave -v -autodefs elisp -index whyse.nw 1>/dev/null
RCS version name $Name:  $
RCS id $Id: noweave.nw,v 1.7 2008/10/06 01:03:24 nr Exp $
(echo @header latex
/usr/local/lib/markup whyse.nw
echo @trailer latex
) |
/usr/local/lib/autodefs.elisp |
/usr/local/lib/finduses |
/usr/local/lib/noidx |
/usr/local/lib/totex
```

Ergo, the simplified pipeline—using Emacs Lisp autodefinitions provided in KNOWEB (written by JOSEPH S. RIEL)—is as follows:

```
markup whyse.nw | autodefs.elisp | finduses | noidx
```

4.0.1 In-development

For an existing project (during development, that is WHYSE) to be loaded, it must minimally be:

1. Parsed, then stored in a database
2. Navigable with WHYSE
   (a) Frame and Windows
   (b) Navigation buttons... at least for modules

This means diagramming the database schema, creating it in EmacSQL, creating validating functions for existing databases, exceptions for malformed databases, and documenting that in LaTeX.

Navigation with WHYSE is multi-part:

1. Query the database for a list of modules, and
2. Create a buffer for the text content retrieved

Exporting a project from the database and editing the project in an in-memory state are further objectives, but they will be archived after the above two have been implemented in a basic form.

4.0.2 TODO

The following features need to be implemented:

1. Project export from database to Noweb format
2. Editing of modules, documentation, and Awk code
3. Navigation with indices
4. Implement indices widgets

5 Parsing

This section covers the parsing of the Noweb tool syntax produced by a project shell script (described in ğ1). The following blocks of LISP code use the PEG Emacs Lisp package to provide for automatic parser generation from a formal PEG grammar based off of the exhaustive description given in the Noweb Hacker's Guide.

5.1 PEG rules

Every character of an input text to be parsed by parsing expressions in a PEG must be defined in terminal rules of the formal grammar. The root rule in the grammar for Noweb tool syntax is the appropriately named `noweb` rule. Beginning `with-peg-rules` brought into scope, the root rule `noweb` is ran on the buffer containing the tool syntax produced by the project shell script.

12a ⟨*buffer parsing function* 12a⟩≡ (23a 26)

```
;;;; Parsing expression grammar (PEG) rules
(defun w-parse-current-buffer-with-rules ()
  "Parse the current buffer with the PEG defined for Noweb tool syntax."
  (with-peg-rules
      (⟨PEG rules 12b⟩)
    (let (w-peg-parser-within-codep)
      (peg-run
       (peg noweb)
       (lambda (lst)
         (setq w-parse-success nil)
         (pop-to-buffer (with-current-buffer (generate-new-buffer "<WHYSE Parse failure log>")
            (insert (format "PEXes which failed:\n%S" lst))
            (current-buffer))))))))
```

Defines:
`w-parse-current-buffer-with-rules`, used in chunks 6a and 26.
Uses `w-parse-success` 1 26.

The grammar can be broken into five sections, each covering some part of parsing.

12b ⟨*PEG rules* 12b⟩≡ (12a)

⟨*high-level Noweb tool syntax structure* 13a⟩
⟨*files and their paths* 13c⟩
⟨*chunks and their boundaries* 14⟩
⟨*quotations* 16e⟩
⟨*keyword definitions* 17a⟩
⟨*meta rules* 13b⟩

As stated, the noweb rule defines the root expression, or starting expression, for the grammar. The tool syntax of Noweb is simply a list of one or more files, which are each composed of at least one chunk. Ergo, the following ⟨*high-level Noweb tool syntax structure* 13a⟩ is defined.

13a ⟨*high-level Noweb tool syntax structure* 13a⟩≡ (12b)
```
;;; Overall Noweb structure
(noweb (bob)
       (not header)
       (+ (and file (* nwnl)
               (or (and x-chunks i-identifiers)
                   (and i-identifiers x-chunks))))
       ;; Trailing documentation chunk and new-lines
       (opt chunk)
       (opt (+ nl))
       (not trailer)

       ;;; DONE: whilst the "no merge error method for (guard t)" issue
       ;;; persists, removing this check for the EOB resolves the issue
       ;;; and lets my own calls to 'error' surface. The issue seems to
       ;;; arise from the fact that I did not handle detection of being
       ;;; within a code chunk properly.
       (eob))
```

It is a fatal error for WHYSE if the header or trailer wrapper keywords appear in the text it is to parse. They are totally irrelevant, and only matter for the final back-ends (TEX, LATEX, or HTML).

The grammar needs to address the fact that the syntax of the Noweb tool format is highly line-oriented, given the influence of AWK on the design (and usage) of Noweb. The following ⟨*meta rules* 13b⟩ define rules which organize the constructs of a line-oriented, or data-oriented, syntax.

13b ⟨*meta rules* 13b⟩≡ (12b)
```
;; Helpers
(nl (eol) "\n")
(!eol (+ (not "\n") (any)))
(spc " ")
```

With the ⟨*meta rules* 13b⟩ enabling easier definitions of what a given "keyword" looks like, the concept of a file needs to be defined. A file is anything that *looks like a file* to Noweb, however, by default only the «*» chunk is tangled when no specific root chunk is given on the command line.

13c ⟨*files and their paths* 13c⟩≡ (12b)
```
;; Technically, file is a tagging keyword, but that classification only
;; makes sense in the Hacker's guide, not in the syntax.
(file (bol) "@file" spc (substring path) nl
      (list (+ chunk))
      `(path chunk-list - (list path chunk-list)))
(path (opt (or ".." ".")) (* path-component) file-name)
(path-component (and path-separator (+ [word])))
(path-separator ["\\/"])
(file-name (+ (or [word] ".")))
```

Because chunks must not overlap, but can nest, the beginnings of chunks need to be pushed to the parsing stack and the end of a chunk needs to be popped off of it. The stack pushing operations in `kind` and `ordinal` delimit chunks by their kinds and number, and the stack actions in the `end` rule check that the chunk-related tokens on the stack are balanced.

14 ⟨*chunks and their boundaries* 14⟩≡ (12b) 15a▷

```
(chunk begin (list (* chunk-contents)) end)
(begin (bol) "@begin" spc kind
       ;; (action (message "A chunk was entered; kind: %s" (cl-first peg-stack)))
       spc ordinal (eol) nl
       (action (if (string= (cl-second peg-stack) "code")
                   (setq w-peg-parser-within-codep t))))
(end (bol) "@end" spc kind
     ;; (action (message "A chunk was exited; kind: %s" (cl-first peg-stack)))
     spc ordinal (eol) nl
     (action (setq w-peg-parser-within-codep nil))
     `(kind-one ordinal-one keywords kind-two ordinal-two -
                (if (and (= ordinal-one ordinal-two) (string= kind-one kind-two))
                            ;;; Push the contents of the chunk to the stack in a cons
                            ;;; cell with the car being a list of the kind and number.
                            ;;;; E.g.:
                    ;; (("code" 3) . (@text @nl @text @nl))
                    (cons (cons kind-one ordinal-one) keywords)
                  (error "There was an issue with unbalanced or improperly nested chunks."))))
(ordinal (substring [0-9] (* [0-9]))
         `(number - (string-to-number number)))
(kind (substring (or "code" "docs")))
```

Writing PEXes for matching file names was the most difficult part I have encountered so far, as it has forced me to understand that a first reading of documentation is usually not sufficient to understand a complex library in an area of programming I have not practiced in before (language parsing).

Valid `chunk-contents` is somewhat confusing, because chunks can contain many types of information other than text and new lines. The definition of what is valid follows.

1. `text`
2. `nl`
3. `defn` *name*
4. `use` *name*
5. `line` *n*
6. `language` *language*
7. `index ...`
8. `xref ...`

Any other keywords are invalid inside a code block. An example of an invalid keyword is anything related to quotations! *This restriction only applies to code blocks, however, and documentation chunks may contain quotations, of course.*

15a ⟨*chunks and their boundaries* 14⟩+≡ (12b) ◁14 16c▷

```
(chunk-contents
 (or
  ⟨structural keywords 15c⟩
  ⟨tagging keywords 16a⟩
  x-notused
  ⟨tool errors 16b⟩))
```

It is easier to handle the fatal keyword appearing inside chunks when it is a permissible keyword to appear inside a chunk; this allows the parser to consider a chunk with fatal inside of it *as a valid chunk*, but that does not mean that a chunk with a fatal keyword inside it does not invalidate a Noweb, it still does: the fatal keyword causes a fatal crash in parsing regardless. Those structural keywords which may be used inside the contents of a chunk are given next.

15b ⟨*structural keywords (except quotations)* 15b⟩≡ (15c)

```
;; structural
text
nwnl ;; Noweb's @nl keyword, as differentiated from the rule nl := "\n".
defn
use ;; NOTE: related to the 'identifier-used-in-module' table.
```

All structural keywords, then, are:

15c ⟨*structural keywords* 15c⟩≡ (15a)

```
⟨structural keywords (except quotations) 15b⟩
quotation
```

16a ⟨*tagging keywords* 16a⟩≡ (15a)
```
;; tagging
line
language
;; index
i-define-or-use
i-definitions
;; xref
x-prev-or-next-def
x-continued-definitions-of-the-current-chunk
i-usages
x-usages
x-label
x-ref
```

16b ⟨*tool errors* 16b⟩≡ (15a)
```
;; error
fatal
```

The fundamental keywords are text and nwnl (new line, per Noweb convention). Text keywords contain source text, and any new lines in the source text are replaced with the appropriate number of nwnl keywords (per convention).

16c ⟨*chunks and their boundaries* 14⟩+≡ (12b) ◁15a 16d▷
```
(text (bol) "@text" spc (substring (* (and (not "\n") (any)))) nl)
(nwnl (bol) "@nl" nl)
```

Nowebs are built from chunks, so the definition and usage of (references to) a chunk are important keywords.

16d ⟨*chunks and their boundaries* 14⟩+≡ (12b) ◁16c
```
(defn "@defn" spc (substring !eol) nl
  ‘(name - (cons "chunk" name)))

(use (bol) "@use" spc (substring !eol) nl
     ‘(chunk-name - (if chunk-name
                        (cons "Chunk usage (child)" chunk-name)
                      (error "UH-OH! There’s a syntax error in the tool output!"))))
```

Documentation may contain text and newlines, represented by @text and [@nwnl]. It may also contain quoted code bracketed by @quote . . . @endquote. Every @quote must be terminated by an @endquote within the same chunk. Quoted code corresponds to the construct in the noweb source.

16e ⟨*quotations* 16e⟩≡ (12b)
```
(quotation (bol) "@quote" nl
           (action (when w-peg-parser-within-codep
                     (error "The parser found a quotation within a code chunk. A @fatal should have been
           (substring (+ (and (not "@endquote") (any))))
           ;; (list (* (or text nwnl defn use i-define-or-use x-ref)))
           (bol) "@endquote" nl
           ‘(lst - (cons "Quotation" lst)))
```

17a ⟨*keyword definitions* 17a⟩≡ (12b) 17b▷
```
(line (bol) "@line" spc (substring ordinal) nl
      `(o - (cons "@line" o)))

(language (bol) "@language" spc (substring words-eol))
```

The indexing and cross-referencing abilities of Noweb are excellent features which enable a reader to navigate through a printed (off-line) or on-line version of the literate document quite nicely. These functionalities each begin with a rule which matches only part of a line of the tool syntax since there are many indexing and cross-referencing keywords. The common part of each line is a rule which merely matches the `@index` or `@xref` keyword. The rest of the lines are handled by a list of rules in `index-keyword` or `xref-keyword`.

The *Noweb Hacker's Guide* lists these two lines in the "Tagging keywords" table, indicating that it's unlikely (or forbidden) that the index or xref keywords would appear alone without any subsequent information on the same line.

> @index ... Index information.
> @xref ... Cross-reference information

There are many keywords defined by the Noweb tool syntax, so they are referenced in this block and defined and documented separately. Some of these keywords are delimiters, so they are not given full "keyword" status (defined as a PEX rule) but exist as constants in the definition of a rule that defines the grouping.

17b ⟨*keyword definitions* 17a⟩+≡ (12b) ◁17a
```
;; Index
⟨indexing and cross-referencing set-off words 17c⟩
⟨fundamental indexing keywords, which are restricted to within a code chunk 18a⟩
⟨the index of identifiers 18d⟩
⟨unsupported indexing keywords 19a⟩

;; Cross-reference
⟨cross-referencing keywords 19b⟩

;; Error
⟨error-causing keywords 20a⟩
```

Further keywords are categorized neatly as Indexing or Cross–referencing keywords, so they are contained in subsections.

5.2 indexing

Indexing keywords, both those used within chunks and those used outside of chunks, are defined in this section. The `«fundamental indexing keywords, which are restricted to within a code chunk»`, index definitions or usages of identifiers and track the definitions of identifiers in a chunk and the usages of identifiers in a chunk. They may seem redundant, but are not; the Noweb Hacker's Guide offers a better explanation of the differences.

17c ⟨*indexing and cross-referencing set-off words* 17c⟩≡ (17b)
```
(idx (bol) "@index" spc)
(xr (bol) "@xref" spc)
```

18a ⟨*fundamental indexing keywords, which are restricted to within a code chunk* 18a⟩≡ (17b)
```
(i-define-or-use
 idx
 (substring (or "defn" "use")) spc (substring !eol) nl
 (action
  (unless w-peg-parser-within-codep
      (error "WHYSE parse error: index definition or index usage occurred outside of a code chunk.")))
 `(s1 s2 - (cons s1 s2)))
```

⟨*identifiers defined in a chunk* 18b⟩
⟨*identifiers used in a chunk* 18c⟩

18b ⟨*identifiers defined in a chunk* 18b⟩≡ (18a)
```
(i-definitions idx "begindefs" nl
               (list (+ (and (+ i-isused) i-defitem)))
               idx "enddefs" nl
               `(definitions - (cons "definitions" definitions)))
(i-isused idx (substring "isused") spc (substring label) nl
          `(u l - (cons u l)))
(i-defitem idx (substring "defitem") spc (substring !eol) nl
           `(d i - (cons d i)))
```

18c ⟨*identifiers used in a chunk* 18c⟩≡ (18a)
```
(i-usages idx "beginuses" nl
          (list (+ (and (+ i-isdefined) i-useitem)))
          idx "enduses" nl
          `(usages - (cons "usages" usages)))
(i-isdefined idx (substring "isdefined" spc label) nl)
(i-useitem idx (substring "useitem" spc !eol) nl) ;; !eol :== ident
```

The summary index of identifiers is a file–specific set of keywords. The index lists all identifiers defined in the file (at least all of those recognized by the autodefinitions filter).

18d ⟨*the index of identifiers* 18d⟩≡ (17b)
```
(i-identifiers idx "beginindex" nl
               (list (+ i-entry))
               idx "endindex" nl)
(i-entry idx "entrybegin" spc (substring label spc !eol) nl
         (list (+ (or i-entrydefn i-entryuse)))
         idx "entryend" nl
         `(e l - (cons e l)))
(i-entrydefn idx (substring "entrydefn") spc (substring label) nl
             `(d l - (cons d l)))
(i-entryuse idx (substring "entryuse") spc (substring label) nl
            `(u l - (cons u l)))
```

The following chunk's name is documentation enough for the purposes of WHYSE. See the Noweb Hacker's Guide for more information.

19a ⟨*unsupported indexing keywords* 19a⟩≡ (17b)
```
;; @index nl was deprecated in Noweb 2.10, and @index localdefn is not
;; widely used (assumedly) nor well-documented, so it is unsupported by
;; WHYSE (contributions for improved support are welcomed).
(i-localdefn idx "localdefn" spc !eol nl)
(i-nl idx "nl" spc !eol nl (action (error ⟨index nl error message 20b⟩)))
```

5.3 cross referencing

19b ⟨*cross-referencing keywords* 19b⟩≡ (17b)
```
(x-label xr (substring "label" spc label) nl)
(x-ref xr (substring "ref" spc label) nl)

(x-prev-or-next-def
 xr (substring (or "nextdef" "prevdef")) spc (substring label) nl
 `(chunk-defn label - (append chunk-defn label)))

(x-continued-definitions-of-the-current-chunk
 xr "begindefs" nl
 (list (+ (and xr (substring "defitem") spc (substring label) nl)))
 ;; NOTE: development statement only; remove this before release.
 ;; (action (message "peg-stack := \n%S" peg-stack))
 xr "enddefs" nl)

(x-usages
 xr "beginuses" nl
 (list (+ (and xr "useitem" spc (substring label) nl)))
 xr "enduses" nl)

(x-notused xr "notused" spc (substring !eol) nl
           `(chunk-name - (cons "notused" chunk-name)))

(x-chunks xr "beginchunks" nl
          (+ x-chunk)
          xr "endchunks" nl)
(x-chunk xr "chunkbegin" spc (substring label) spc (substring !eol) nl
         (list (+ (list (and xr
                             (substring (or "chunkuse" "chunkdefn"))
                             spc
                             (substring label)
                             nl))))
         xr "chunkend" nl)

;; Associates label with tag (@xref tag $LABEL $TAG)
(x-tag xr "tag" spc label spc !eol nl)
(label (+ (or "-" [alnum]))) ;; A label never contains whitespace.
```

20a ⟨*error-causing keywords* 20a⟩≡ (17b)
```
;; User-errors (header and trailer) and tool-error (fatal)
;; Header and trailer's further text is irrelevant for parsing, because they cause errors.
(header (bol) "@header" ;; formatter options
        (action (error "[ERROR] Do not use totex or tohtml in your noweave pipeline.")))
(trailer (bol) "@trailer" ;; formatter
         (action (error "[ERROR] Do not use totex or tohtml in your noweave pipeline.")))
(fatal (bol) "@fatal"
       (action (error "[FATAL] There was a fatal error in the pipeline. Stash the work area and submit a
```

20b ⟨*index nl error message* 20b⟩≡ (19a)
```
(string-join
 '("\"@index nl\" detected."
  "This indicates hand-written @ %def syntax in the Noweb source."
  "This syntax was deprecated in Noweb 2.10, and is entirely unsupported."
  "Write an autodefs AWK script for the language you are using.")
 "\n")
```

6 Packaging

Installing an Emacs Lisp package is quite easy if the system is distributed through the GNU Emacs Lisp Package Archive (GNU ELPA), and only slightly less easy if it is distributed through MELPA (Milkypostman's Emacs Lisp Package Archive). Other package archives have existed, but they are all ephemeral. The most popular alternative to GNU ELPA, Non-GNU ELPA, and MELPA is direct distribution of files through Git servers and the use of a package by the end user to install directly from such.

This software is in-development, so it will only be distributed directly through Git.

WHYSE follows the form of "simple", single-file packages documented in the Emacs Lisp Reference Manual. The package file, `whyse.el`, is emitted by `notangle` which is called by the Makefile in every target but `clean`. All source development occurs in `whyse.nw` using POLYMODE.

The makefile distributed alongside whyse.nw in the tarball contains the command-line used to tangle and weave WHYSE.

20c ⟨*whyse.el* 20c⟩≡
- ⟨*Emacs Lisp package headers* 21a⟩
- ⟨*Licensing and copyright* 22a⟩
- ⟨*Commentary* 22b⟩
- ⟨*Code* 23a⟩
- ⟨*EOF* 23b⟩

21a ⟨*Emacs Lisp package headers* 21a⟩≡ (20c)

```
;;; whyse.el -- noWeb HYpertext System in Emacs -*- lexical-binding: t -*-

;; Copyright ℗ 2023 Bryce Carson

;; Author: Bryce Carson <bcars268@mtroyal.ca>
;; Created 2023-06-18
;; Keywords: tools tex hypermedia
;; URL: https://cyberscientist.ca/whyse

;; This file is not part of GNU Emacs.
```

Uses whyse 1 4b.

21b ⟨*whyse-pkg.el* 21b⟩≡

```
(define-package "whyse" "0.1" "noWeb HYpertext System in Emacs"
  '(⟨required packages 21c⟩))
```

Uses whyse 1 4b.

The Emacs Lisp Manual states, regarding the `Package-Requires` element of an Emacs Lisp package header:

> Its format is a list of lists on a single line.

Thus, to prevent spill–over in the printed document, the ⟨*required packages* 21c⟩ are given on separate lines in the literate document. When the file is tangled, however, a Noweb filter will be used to ensure that all required packages are on a single line by simply removing the new lines from the following code chunk. The same principle is followed for the ⟨*file-local variables* 23c⟩.

21c ⟨*required packages* 21c⟩≡ (21b)

```
(emacs "25.1")
(emacsql "20230220")
(dash "20230617")
(peg "1.0.1")
(cl-lib "1.0")
(ts "20220822")
```

22a ⟨*Licensing and copyright* 22a⟩≡ (20c)

```
;; This program is free software: you can redistribute it and/or
;; modify it under the terms of the GNU General Public License as
;; published by the Free Software Foundation, either version 3 of the
;; License, or (at your option) any later version.

;; This program is distributed in the hope that it will be useful, but
;; WITHOUT ANY WARRANTY; without even the implied warranty of
;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
;; General Public License for more details.

;; You should have received a copy of the GNU General Public License
;; along with this program. If not, see
;; <https://www.gnu.org/licenses/>.

;; If you cannot contact the author by electronic mail at the address
;; provided in the author field above, you may address mail to be
;; delivered to

;; Bryce Carson
;; Research Assistant
;; Dept. of Biology

;; Mount Royal University
;; 4825 Mount Royal Gate SW
;; Calgary, Alberta, Canada
;; T3E 6K6
```

22b ⟨*Commentary* 22b⟩≡ (20c)

```
;;; Commentary:
;; WHYSE was described by Brown and Czedjo in _A Hypertext for Literate
;; Programming_ (1991).
;;
;; Brown, M., Czejdo, B. (1991). A hypertext for literate programming.
;;    In: Akl, S.G., Fiala, F., Koczkodaj, W.W. (eds) Advances in
;;    Computing and Information  ICCI '90. ICCI 1990. Lecture Notes in
;;    Computer Science, vol 468. Springer, Berlin, Heidelberg.
;;    https://doi-org.libproxy.mtroyal.ca/10.1007/3-540-53504-7_82.
;;
;; A paper describing this implementation--written in Noweb and browsable,
;; editable, and auditable with WHYSE, or readable in the printed form--is
;; hoped to be submitted to The Journal of Open Source Software (JOSS)
;; before the year 2024. N.B.: the paper will include historical
;; information about literate programming, and citations (especially
;; of those given credit here for ideating WHYSE itself).
```

23a ⟨*Code* 23a⟩≡ (20c)

```
;;; Code:
;;;; Compiler directives
(eval-when-compile (require 'wid-edit))

;;;; Internals
⟨Customization and global variables 1⟩
⟨Widgets 3⟩
⟨WHYSE project structure 5b⟩
⟨buffer parsing function 12a⟩

;;;; Commands
;;;###autoload
⟨WHYSE 4b⟩
```

23b ⟨*EOF* 23b⟩≡ (20c)

```
(provide 'whyse)

⟨file-local variables 23c⟩
```

23c ⟨*file-local variables* 23c⟩≡ (23b)

```
;; Local Variables:
;; mode: emacs-lisp
;; no-byte-compile: t
;; no-native-compile: t
;; End:
```

7 Indices

7.1 Chunks

⟨*Widgets* 3⟩

7.2 Identifiers

Underlined indices denote definitions; regular indices denote uses.

8 Appendices

8.1 A user-suggested functionality: `w-with-project`

It was suggested during early development that ⟨*API-like functions* 25⟩ such as `w-with-project` be written. An early version of such functionality is provided in `w-with-project`.

25 ⟨*API-like functions* 25⟩≡

```
;; This chunk intentionally left blank at this time.
```

9 Tests

TODO: adopt the ERT (Emacs Regression Tests) package to test WHYSE features as they are developed and become featureful. When a feature is implemented a test should be written which conforms to the current documentation so that regressions can be caught when changes are made.

9.1 Parsing Tests

9.1.1 Parsing tool syntax within a temporary buffer

26 ⟨*test-parser-with-temporary-buffer.el* 26⟩≡

```
(defvar w-parse-success t
  "A simple boolean regarding the success or fialure of the last attempt to parse a buffer of Noweb tool

⟨buffer parsing function 12a⟩

(with-temp-buffer
  (insert (shell-command-to-string
           "make -silent -file ~/src/whs/Makefile tool-syntax"))
  (goto-char (point-min))
  (w-parse-current-buffer-with-rules)
  ;; (pop-to-buffer
  ;;  (clone-buffer
  ;;   (generate-new-buffer-name
  ;;    (format "<WHYSE %s> Parsing tool syntax with a temporary buffer"
  ;;            (if w-parse-success "SUCCESS" "FAILURE")))))
  )

;; Local Variables:
;; mode: lisp-interaction
;; no-byte-compile: t
;; no-native-compile: t
;; eval: (read-only-mode)
;; End:
```

Defines:
`w-parse-success`, used in chunk 12a.
Uses `w-parse-current-buffer-with-rules` 12a.