1. WHYSE PROJECTS

The organization of this literate program is linear, with aspects of the program explained as the user would encounter them, more or less. A user will read from the package description that they should call an interactive command to create a project. The WHYSE. application has a single interactive command: `whyse`. The command loads the first element of the customization variable `w-registered-projects`, considering that the default project, or it opens the "Easy Customization Interface" for the application's customization group (`M-x customize-group whyse`): an effective prompt for the user to enter the necessary information. If user's dislike this, they can disable it.

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.

§??? <<Customization and global variables ???>>≡

This definition continued in ??? and ???.

This code used in ???.

```
(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")
```

Defines:
`w-registered-projects`, not used in this document.
`whyse`, not used in this document.

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."

§??? <<Widgets ???>>≡

This code used in ???.

```
(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 ""))))
```

An example of what the list generated from the information entered into Customize would look like is given here for elucidation (as it would exist in a `custom-set-variables` form).

```
'(w-registered-projects     '(("noWeb HYpertext System in Emacs"           "~/Desktop/whyse.nw"           "make  -C ~/Desktop  --silent  --file  ~/src/whyse/Makefile  tool-syntax"))     nil     (widget))
```

The function documentation string should be expalnatory enough for the behaviour of the `whyse` command.

§??? <<WHYSE ???>>≡

This code used in ???.

```
(defun whyse ()
  "Opens the default whyse project, conditionally running hooks.

Hooks are only run if a project is actually opened. If
‘w-load-default-project?’ and
‘w-open-customize-when-no-project-defined?’ are both nil then a
warning is given and hooks are not run.

When both customization variables are non-nil, or if only
‘w-load-default-project?’ is nil, then Customize is opened to the
whyse group."
  (interactive)
  ;; Warn the user that their customization options have made ‘whyse’ a
  ;; no-op function.
  (when (and (not w-load-default-project?)
             (not w-open-customize-when-no-projects-defined?))
    (warn "The customization options for ‘whyse’ have effectively disabled the ‘whyse’ command."))
  (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)))
           (parse-tree (w-parse-with-project-and-temp-buffer project)))

      ;; FIXME: peculiar error: "UNIQUE constraint failed:
      ;; module.module_number", 19, nil, "constraint failed"
      (progn
        <<setup project database ???>>
        (run-hooks 'w-open-project-hook))
    (unless (not w-open-customize-when-no-projects-defined?)
      (customize-group 'whyse))))
```

Defines:
  `whyse`, not used in this document.
Uses `w-load-default-project?` ???, `w-open-customize-when-no-projects-defined?` ???, `w-parse-with-project-and-temp-buffer` ??? and `w-registered-projects` ???.

§??? <<Customization and global variables ???>>+≡

```
(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?")

(defcustom w-open-customize-when-no-projects-defined? t
  "Non-nil values mean the system will open Customize as
necessary.

nil will cause ‘whyse’ to simply do nothing when no project is
defined."
  :type 'boolean
  :group 'whyse
  :tag "Open Customize to the whyse group when ‘whyse’ is
                           invoked and no projects are defined?")
```

Defines:
  `w-load-default-project?`, not used in this document.
  `w-open-customize-when-no-projects-defined?`, not used in this document.
Uses `whyse` ???.

The structure accessed in the namesake command of the package is rather simple.

It is defined quickly, then explained briefly.

§??? <<WHYSE project structure ???>>≡

This code used in ???.

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

 ;; Usage
 frame

 ;; Metadata
 (date-created (current-time-string))
 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-open-project-hook` is run.

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.

§??? <<return a filename for the project database ???>>≡

This code used in ???.

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

For 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.

§??? <<setup project database ???>>≡

This code used in ???.

```
<<create a database connection ???>>
<<map over SQL s-expressions, creating the tables ???>>
```

§??? <<create a database connection ???>>≡

This code used in ???.

```
(setf (w-project-database-connection project)
    (emacsql-sqlite <<return a filename for the project database ???>>))
```

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

§??? <<map over SQL s-expressions, creating the tables ???>>≡

This code used in ???.

```
(mapcar (lambda (expression)
       (emacsql (w-project-database-connection project) expression))

    ;; A list of SQL s-expressions to create the tables.
      '([:create-table-if-not-exists module
       ([module-name
         content
         file-name
         section-name
         (displacement integer)
         (module-number integer :primary-key)])]

      [:create-table-if-not-exists parent-child
       ([(parent integer) (child  integer) (line-number integer)]
        (:primary-key [parent child]))]

      [:create-table-if-not-exists 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-if-not-exists topic-referenced-in-module
       ([(topic-name nil)
         (module-number integer)]
        (:primary-key [topic-name module-number]))]))
```

3. Customizing the behaviour of

`whyse` with hooks"

WHYSE is meant to be customizable, defining as little as necessary to "implement a development environment for Noweb as described by Brown and" Czejdo (TODO{cite these again}).

§??? <<open-project-hook ???>>≡

This code used in ???.

```
(defvar w-open-project-hook '()
  "Hooks to run when ‘whyse’ has opened a project.")
```

Defines:
`w-open-project-hook`, not used in this document.

The default behaviour of WHYSE. is to insert all the chunks of the parsed document into a database. Before it does that it works upon the parse tree, preparing it into a suitable format usable with EmacSQL (which the author is aware he’s stated elsewhere).

§??? <<default hook functions ???>>≡

This code used in ???.

```
(defun w--log-in-buffer (buffer-name &rest body)
  "In a new buffer named BUFFER-NAME, insert the value of evaluating BODY."
  (save-mark-and-excursion
    (with-current-buffer
        (generate-new-buffer buffer-name)
      (insert (format-message "%S" body)))))

(defun w--prepare-sexp-sql-from-file-tokens ()
  "Prepare an s-expression of SQL statements for ‘emacsql’.

This hook depends on this object being in scope: ‘parse-tree’.
That object is in scope when this hook runs with the default
implementation of ‘whyse’ (this is not meant to imply there are
non-default implementations, only that hacked up installs won’t
operate with any guarantees)."
  (mapcar
   (lambda (file-token)
     (let ((file-name (car file-token))
           (chunks (cdr file-token)))
       (emacsql (w-project-database-connection project)
                (vector :insert :into 'module
                        :values (mapcar (lambda (chunk)
                                          "Convert an individual chunk to a vector of objects."
                                          ;; If the chunk has a name, it is a code chunk;
                                          ;; otherwise it’s documentation.
                                          (vector (w--chunk-name chunk)
                                                  (w--chunk-text chunk)
                                                  file-name
                                                  nil
                                                  nil
                                                  (w--chunk-number chunk)))
                                        ;; An illustration of the structure of ‘chunks’:
                                        ;; (list '((a . 1) (b . 2) (c . 3))
                                        ;;       '((a . 1) (b . 2) (c . 3)))
                                        chunks)))))
   parse-tree))

(add-hook 'w-open-project-hook 'w--prepare-sexp-sql-from-file-tokens)
```

Defines:
`w--log-in-buffer`, not used in this document.
`w--prepare-sexp-sql-from-file-tokens`, not used in this document.
Uses `w--chunk-name` ???, `w--chunk-number` ??? and `w--chunk-text` ???.

4. Parsing project nowebs

This section covers the parsing of the noweb tool syntax produced when `whyse` executes the project’s defined shell script to generate the tool syntax.

The package provides automatic parser generation from a formal PEG. grammar. The grammar is based off of the description of the tool syntax given in the Noweb Hacker’s Guide. Guide}

5. 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.

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

§??? <<PEG rules ???>>≡

This code used in ???.

```
<<high-level Noweb tool syntax structure ???>>
<<files and their paths ???>>
<<chunks and their boundaries ???>>
<<quotations ???>>
<<keyword definitions ???>>
<<meta rules ???>>
```

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 ???>> is defined.

§??? <<high-level Noweb tool syntax structure ???>>≡

This code used in ???.

```
;;; Overall Noweb structure
(noweb (bob) (not header) (+ file) (not trailer) (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, eX, or HTML) that produce human-readable documenation.

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 (a historical version was entirely implemented in AWK). The following <<meta rules ???>> define rules which organize the constructs of a line-oriented, or data-oriented, syntax.

§??? <<meta rules ???>>≡

This code used in ???.

```
;; Helpers
(nl (eol) "\n")
(!eol (+ (not "\n") (any)))
(spc " ")
```

With the <<meta rules ???>> 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 named ‘‘*’’ (it’s chunk header is `<<*>>`) is tangled when no specific root chunk is given on the command line.

§??? <<files and their paths ???>>≡

This code used in ???.

```
;; 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 (and (+ chunk)
               (list (or (and x-chunks i-identifiers)
                         (and i-identifiers x-chunks))))
          ;; Trailing documentation chunk and new-lines after the xref
          ;; and index.
          (opt chunk)
          (opt (+ nl)))
    '(path chunk-list -- (cons 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.

§??? <<chunks and their boundaries ???>>≡

This definition continued in ???, ??? and ???.

This code used in ???.

```
(chunk begin (list (* chunk-contents)) end)
(begin (bol) "@begin" spc kind spc ordinal (eol) nl
    (action (if (string= (cl-second peg--stack) "code")
                (setq w--peg-parser-within-codep t))))
(end (bol) "@end" spc kind spc ordinal (eol) nl
   (action
    (setq w--peg-parser-within-codep nil))
   ;; The stack grows down and the heap grows up,
   ;; that's the yin and yang of the computer thang
   '(kind-one
     ordinal-one
     keywords
     kind-two
     ordinal-two
     --
     (if (and (= ordinal-one ordinal-two) (string= kind-one kind-two))
         (cons (cons (if (string= kind-one "code")
                         'code
                       'docs)
                     ordinal-one)
               keywords)
       (error "Chunk nesting error encountered."))))
(ordinal (substring [0-9] (* [0-9]))
      '(number -- (string-to-number number)))
(kind (substring (or "code" "docs")))
```

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. *name name n language*

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. As an exception, the keywords were originally banned inside code chunks, but to parse the noweb document in which WHYSE itself was written it needed to be adjusted. The grammar should be studied again to ensure that textual description and reality are in step.

§??? <<chunks and their boundaries ???>>+≡
```
(chunk-contents
 (or
  <<structural keywords ???>>
  <<tagging keywords ???>>
  x-notused
  <<tool errors ???>>))
```

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.

§??? <<structural keywords (except quotations) ???>>≡

This code used in ???.

```
;; 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:

§??? <<structural keywords ???>>≡

This code used in ???.

```
<<structural keywords (except quotations) ???>>
quotation
```

and no more.

There are tagging keywords also, which are used to associate metadata with certain texts.

§??? <<tagging keywords ???>>≡

This code used in ???.

```
;; 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
```

TODO: Verify that this statement is true: "Usually Noweb will warn a user that a chunk was referenced but undefined, or that there was some other issue with chunks." Sometimes, however, the system will permit a chunk to be undefined and this leads to the only cases in the tool syntax where it is not line-oriented. `noidx` will read the cross references to other chunks and will be unable to generate the label, so it will insert `@notdef` where it would otherwise upcase ‘‘nw’’ and then

insert the label. This is why `x-undefined` is placed among the other <<tool errors ???>> keywords.

§??? <<tool errors ???>>≡

This code used in ???.

```
;; error
fatal
x-undefined
```

The fundamental keywords are text and nwnl (new line, per Noweb convention). Text keywords contain source text, and any new line tokens in the source text are replaced with the appropriate number of @nl keywords (per convention); these are reduced to a single text token when they are adjacent on the `peg--stack`.

§??? <<chunks and their boundaries ???>>+≡

```
(text (bol) "@text" spc (substring (* (and (not "\n") (any))))
nl
      '(txt -- (w--concatenate-text-tokens (cons 'text txt))))
(nwnl (bol) (substring "@nl") nl
      ;; Be sure that when thinking about the symbol 'nl' here
that
      ;; you're not confusing it with the peg rule nl.
      '(nl -- (w--concatenate-text-tokens (cons 'nl "\n"))))
```

Uses `w--concatenate-text-tokens` ???.

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

§??? <<chunks and their boundaries ???>>+≡

```
(defn "@defn" spc (substring !eol) nl
  '(name -- (cons 'chunk name)))

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

*[pg*header]
*[pg*odd-header]