The Implementation of WHYSE

Bryce Carson

April 29, 2024

Contents

Abstract

(no)Web HYpertext System in Emacs (WHYSE) is an integrated development environment for Noweb and LaTeX within Emacs, similar to EDE but not sharing development principles. It is based off of an academic paper written in 1991 by Brown and Czejdo. 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 in the `<<Commentary>>` for ideating WHYSE itself).

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). It may, however, include the writing of filters implemented with Sed, AWK, or other languages. Users must know how to write a custom command-line for noweave (read the manual section regarding the `-v` option). If you only know how to call the noweave command you're reading the wrong document. Read the Noweb manual first, please. Developers of WHYSE extensions 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.

0.1 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.

1 ⟨Customization and global variables 1⟩≡

```
(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`, used in 1 2<2 chunks 0 and 0.
`whyse`, used in 1 1<1 1<1 1<1 3<3 1<1 1<1 14<14 1<1 19<19 19<19 1<1 22<22 chunks 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, and 0.

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.

1 ⟨Widgets 1⟩≡

```
(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.

2 ⟨WHYSE 2⟩≡

```
  (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 4⟩
(run-hooks 'w-open-project-hook))
(unless (not w-open-customize-when-no-projects-defined?)
(customize-group 'whyse))))
```

Defines:
`whyse`, used in 1 1<1 1<1 1<1 3<3 1<1 1<1 14<14 1<1 19<19 19<19 1<1 22<22 chunks 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, and 0.
Uses `w-load-default-project?` 3, `w-open-customize-when-no-projects-defined?` 3, `w-parse-with-project-and-temp-buffer` 14, and `w-registered-projects` 1.

3 ⟨Customization and global variables 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?")

(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?`, used in 2 chunk 0.
`w-open-customize-when-no-projects-defined?`, used in 2 chunk 0.
Uses `whyse` 1 2.

The structure accessed in the namesake command of the package is rather simple. TODO Ensure that the previous statement in-prose [not in the TODO summary] is still correct. It is defined quickly, then explained briefly.

3 ⟨WHYSE project structure 3⟩≡

```
(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. TODO Describe initialization of the system after parsing.

0.1.1 Database initialization

TODO finish the creation of a database. Use what I learned in the fall!

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.

4 ⟨return a filename for the project database 4⟩≡

```
(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 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.

4 ⟨setup project database 4⟩≡

```
⟨create a database connection 4⟩
⟨map over SQL s-expressions, creating the tables 5⟩
```

4 ⟨create a database connection 4⟩≡

```
(setf (w-project-database-connection project)
(emacsql-sqlite ⟨return a filename for the project database 4⟩))
```

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

5 ⟨map over SQL s-expressions, creating the tables 5⟩≡
```
(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]))]))
```

0.2 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).

5 ⟨open-project-hook 5⟩≡
```
(defvar w-open-project-hook '()
"Hooks to run when `whyse' has opened a project.")
```
Defines:
`w-open-project-hook`, used in 5 chunk 0.

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).

5 ⟨default hook functions 5⟩≡

```
(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`, never used.
  `w--prepare-sexp-sql-from-file-tokens`, never used.
Uses `w--chunk-name` 17, `w--chunk-number` 17, and `w--chunk-text` 17.

0.3 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 peg 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. TODO formally cite the Noweb Hacker's Guide

0.3.1 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.

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

7 ⟨PEG rules 7⟩≡
⟨high-level Noweb tool syntax structure 7⟩
⟨files and their paths 8⟩
⟨chunks and their boundaries 8⟩
⟨quotations 11⟩
⟨keyword definitions 11⟩
⟨meta rules 7⟩

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 7⟩ is defined.

7 ⟨high-level Noweb tool syntax structure 7⟩≡
```
;;; 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, LATEX, 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 7⟩ define rules which organize the constructs of a line-oriented, or data-oriented, syntax.

7 ⟨meta rules 7⟩≡
```
;; Helpers
(nl (eol) "\n")
(!eol (+ (not "\n") (any)))
(spc " ")
```

TODO: Review the following paragraph and rephrase it.

With the ⟨meta rules 7⟩ 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.

TODO: Write about the need for the overall document to be separate from the one-or-more files specified in the document. Exempli gratia: the current document, contained in `whyse.nw` contains two files, though they are separately tangled: `whyse.el` and `test-parser-with-temporary-buffer.el`. If these two files were tangled at the same time, such that the output file discovery

ability of Noweb was used, then the there would be more than one file in the intermediate tool syntax, but still a single preceeding documentation chunk before the first file, and a single succeeding documentation chunk after the last file.

8 ⟨files and their paths 8⟩≡
```
;; 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] ".")))
```

NOTE: 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).

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.

8 ⟨chunks and their boundaries 8⟩≡
```
(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.

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. 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.

9 ⟨chunks and their boundaries 8⟩+≡

```
(chunk-contents
(or
⟨structural keywords 9⟩
⟨tagging keywords 10⟩
x-notused
⟨tool errors 10⟩))
```

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.

9 ⟨structural keywords (except quotations) 9⟩≡

```
;; 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:

9 ⟨structural keywords 9⟩≡

```
⟨structural keywords (except quotations) 9⟩
quotation
```

10 ⟨tagging keywords 10⟩≡
```
;; 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 10⟩ keywords.

10 ⟨tool errors 10⟩≡
```
;; 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`.

10 ⟨chunks and their boundaries 8⟩+≡
```
(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` 18.

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

10 ⟨chunks and their boundaries 8⟩+≡
```
(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!"))))
```

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.

11 ⟨quotations 11⟩≡
```
(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
found here, but was not.")))
(substring (+ (and (not "@endquote") (any))))
(bol) "@endquote" nl
`(lst -- (cons 'quotation lst)))
```

11 ⟨keyword definitions 11⟩≡
```
(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.

11 ⟨keyword definitions 11⟩+≡
```
;; Index
⟨indexing and cross-referencing set-off words 12⟩
⟨fundamental indexing keywords, which are restricted to within a code chunk 12⟩
⟨the index of identifiers 12⟩
⟨unsupported indexing keywords 13⟩

;; Cross-reference
⟨cross-referencing keywords 13⟩

;; Error
⟨error-causing keywords 14⟩
```

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

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.

12 ⟨indexing and cross-referencing set-off words 12⟩≡
```
(idx (bol) "@index" spc)
(xr (bol) "@xref" spc)
```

12 ⟨fundamental indexing keywords, which are restricted to within a code chunk 12⟩≡
```
(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 (make-symbol s1) s2)))

⟨identifiers defined in a chunk 12⟩
⟨identifiers used in a chunk 12⟩
```

12 ⟨identifiers defined in a chunk 12⟩≡
```
(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 'used! l)))
(i-defitem idx (substring "defitem") spc (substring !eol) nl
`(d i -- (cons 'def-item i)))
```

12 ⟨identifiers used in a chunk 12⟩≡
```
(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).

12 ⟨the index of identifiers 12⟩≡
```
(i-identifiers idx "beginindex" nl
(list (+ i-entry))
idx "endindex" nl
`(l -- (cons 'i-identifiers l)))
```

```
(i-entry idx "entrybegin" spc (substring label spc !eol) nl
(list (+ (or i-entrydefn i-entryuse)))
idx "entryend" nl
`(entry-label lst -- (cons 'entry-label lst)))
(i-entrydefn idx (substring "entrydefn") spc (substring label) nl
`(defn label -- (cons 'defn label)))
(i-entryuse idx (substring "entryuse") spc (substring label) nl
`(use lst -- (cons 'use lst)))
```

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

`@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).

13 ⟨unsupported indexing keywords 13⟩≡

```
;; @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 14⟩)))
```

cross referencing

13 ⟨cross-referencing keywords 13⟩≡

```
(x-label xr (substring "label" spc label) nl
`(substr -- (cons 'x-label (cadr (split-string substr)))))
(x-ref xr (substring "ref" spc label) nl
`(substr --  (cons 'ref (cadr (split-string substr)))))

;; FIXME: improve the error handling at this point. It is not fragile
;; any longer, becasue most things are ignored and this is hackish;
;; however, the message reporting is not too helpful. It would be nice
;; to have _only_ the chunk name reported, and formatted with << and >>.
;;; Reproduction steps: make a reference to an undefined code chunk
;;; within another code chunk. For fixing this issue, undefined code
;;; chunks should also be referenced within quotations in documentation.
(x-undefined
xr (or "ref" "chunkbegin") spc
(guard
(if (string= "nw@notdef"
(buffer-substring-no-properties (point) (+ 9 (point))))
(error (format "%s: %s: %s:\n@<@<%s>>"
"WHYSE"
"nw@notdef detected"
"an undefined chunk was referenced"
(buffer-substring-no-properties (progn (forward-line) (point))
(end-of-line)))))))

(x-prev-or-next-def
xr (substring (or "nextdef" "prevdef")) spc (substring label) nl
`(previous-or-next-chunk-defn label -- (cons (make-symbol previous-or-next-chunk-
defn) label)))

(x-continued-definitions-of-the-current-chunk
xr "begindefs" nl
```

```
(list (+ (and xr (substring "defitem") spc (substring label) nl)))
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
`(name -- (cons 'unused! name)))
(x-chunks nwnl
nwnl
xr "beginchunks" nl
(list (+ x-chunk))
xr "endchunks" nl
`(l -- (cons 'x-chunks l)))
(x-chunk xr "chunkbegin" spc (substring label) spc (substring !eol) nl
(list (+ (list (and xr
(substring (or "chunkuse" "chunkdefn"))
`(chunk-usage-or-definition -- (make-symbol chunk-usage-or-definition))
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.
```

14 ⟨error-causing keywords 14⟩≡

```
;; 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 bug report against Noweb, WHYSE, and other relevant tools.")))
```

14 ⟨index nl error message 14⟩≡

```
(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")
```

To summarize this section, since it is longer than the previous section, the object is to convert the noweb document to tool syntax and parse it with the peg parser.

14 ⟨with-project 14⟩≡

```
(defun w-parse-with-project-and-temp-buffer (project)
"Parses a project PROJECT in a temporary buffer.

PROJECT must be registerd with whyse in the
```

```
  `w-registered-projects' customization variable, and PROJECT is a
  member of that list."
  (with-temp-buffer
  (insert (shell-command-to-string (w-project-script project)))
  (goto-char (point-min))
  (w--parse-current-buffer-with-rules)))
```

Defines:
  w-parse-with-project-and-temp-buffer, used in 2 chunk 0.
Uses w--parse-current-buffer-with-rules 15 and whyse 1 2.

15 ⟨buffer parsing function 15⟩≡

```
  ;; FIXME: the current parse tree contains a `nil' after the chunk type
  ;; and number assoc, and that needs to be analyzed. Why is this `nil' in
  ;; the stack? I assume and believe it is because of the collapsing of
  ;; stringy tokens; when a token should be put back onto the stack it may
  ;; also be putting a `nil' onto the stack in the first call to the
  ;; function.
  ;;;; 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 7⟩)
  (let (w--peg-parser-within-codep
  (w--first-stringy-token? t))
  (peg-run (peg noweb) #'w--parse-failure-function))))

  (defun w--parse-failure-function (lst)
  (setq w--parse-success nil)
  (pop-to-buffer (clone-buffer))
  (save-excursion
  (put-text-property (point) (point-min)
  'face 'success)

  (put-text-property (point) (point-max)
  'face 'error)

  (goto-char (point-max))
  (message "PEXes which failed:\n%S" lst)))
```

Defines:
  w--parse-current-buffer-with-rules, used in 14 22<22 chunks 0 and 0.
  w--parse-failure-function, never used.
Uses w--parse-success 15 22.

15 ⟨Customization and global variables 1⟩+≡

```
  (defvar w--parse-success t
  "The success or failure of the last parsing of noweb tool syntax.")
```

Defines:
  w--parse-success, used in 15 chunk 0.

0.4 Processing parsed nowebs into SQL

This section covers how the parsed text generated in the last section is processed, creating a series of SQL statements that will be executed by SQLite using the interface provided by the EmacSQL package.

First, the overall structure of the parsed text should be diagrammed. The parse tree is a list of noweb documents, each being a list themselves. The first atom of an inner list, corresonding to a document, is the filename of that document (hopefully the same filename as passed on the commmand-line elsewhere when the document is used).

Deeper, each document-list contains as the second atom a list of all of its contents, which is an association list thereof. Each association in the alist should be self-explanatory.

```
'((noweb-document-one
   ((docs . 0)
    (text . "\tex{} is cool!"))
   ((code . 1)
    (text . "(message \"LISP is awesome!\")")))
  (noweb-document-two
   ((code . 0)
    (text . "asdf is a system definition format in Common LISP,"))
    (nwnl . "\n")
    (text . "and I like to use it.")
   ((code . 1)
    (text . "jkl; is the right-handed corollary of asdf."))
   ((docs . 2)
    (text . "\latex{} is great!"))
   ((docs . 3)
    (text . "Noweb, written by Norman Ramsey is sweet!"))))
```

There are many steps to compiling the parse tree into SQL that can be directly executed by the backend database engine. The first step is to ensure the parse tree is in a format that is acceptable to other LISP functions; this will make it easier to navigate the tree and transform it. Other texts call this (list or expression) destructuring.

The first step in making the parse tree navigable for other programs is collapsing adjacent "stringy" tokens into single `text` tokens. The output tool syntax of notangle, and the parse tree resulting from the PEG, contain individual text tokens for fragments of whole text lines and form feed characters. These tokens exist because the cross-referencing tokens fragment the text lines, and new lines in the noweb document are treated specially to facilitate this fragmentation. The parsed from of the tool syntax is shown in this example from a development version of WHYSE.

```
(text . "  and \textsc{Noweb}'s \texttt{finduses.nw}!")
(nwnl . "@nl")
(text . "\end{enumerate}")
(nwnl . "@nl")
(text . "")
(nwnl . "@nl")
```

In this development version it was not fully decided how tokens and the data they correspond to should be arranged, so the newlines are not part of a list, while the text characters are part of an outer plist of which the parentheses are not visible in this example.

To collapse these tokens into a single text token the `peg--stack` must be manipulated carefully. It isn't advisable to manipulate this variable in the course of a PEG grammar's actions, however, there is a use case for it when the previous rules and actions won't accomodate the necessary action without refactoring a larger part of the grammar. In this development version that is not a goal; basic functionality is sought after, not robustness or beauty, so hacking the desired behaviour together quickly is better.

`w--nth-chunk-of-nth-noweb-document` retrieves the parse tree for the nth noweb document, which in the case of `whyse.nw` is the parse tree of the zeroth-indexed document. It's quite a simple function. To obtain a given chunk of this document from the parse tree the result of the function is called with `nth` and the index of the chunk.

17 ⟨functions for navigating WHYSE parse trees 17⟩≡

```
(defun w--nth-document-file-name (nth-document parse-tree)
"Return the file name of the nth-indexed document in the parse tree.

For the first document in the parse tree, that is the
zeroth-indexed document."
(cl-first (nth nth-document parse-tree)))

(defun w--nth-document (nth-document parse-tree)
"Return the subtree of the nth-indexed document in the parse tree."
(cl-second (nth nth-document parse-tree)))

(defun w--nth-chunk-of-document (n document)
"Return the subtree for the Nth chunk of a noweb document parse subtree."
(nth n document))

(defun w--chunk-number (chunk)
"Return the chunk number of CHUNK."
(or (cdr (assq 'code chunk))
(cdr (assq 'docs chunk))))

(defun w--chunk-text (chunk)
"Join all the strings returned from the collection in the loop,
and return the single string."
(string-join
(cl-loop for elt in chunk collect
(when (and (listp elt) (equal 'text (car elt)))
(cdr elt)))
""))

(defun w--chunk-name (chunk)
"Return non-nil if CHUNK is a code chunk, and thereby has a name.

The return value, if non-nil, is actually the name of the chunk."
(if-let ((name (assq 'chunk chunk)))
(cdr name)))
```

Defines:

`w--chunk-name`, used in 5 chunk 0.
`w--chunk-number`, used in 5 chunk 0.
`w--chunk-text`, used in 5 chunk 0.
`w--nth-chunk-of-document`, never used.
`w--nth-document`, never used.
`w--nth-document-file-name`, never used.

TODO: place this inclusion better

18 ⟨Code 18⟩≡
⟨functions for navigating WHYSE parse trees 17⟩
⟨functions to collapse text and newline tokens into their largest possible form 18⟩

18 ⟨functions to collapse text and newline tokens into their largest possible form 18⟩≡
```
(defun w--concatenate-text-tokens (new-token)
"Join the values of two text token associations in a two-element token alist.

If the two associations shouldn't be joined, return them to the stack."
(prog1
;; Concatenation only occurs when the previous token examined was
;; a text or nwnl token, ergo there must have been a text or nwnl
;; token previously examined for any concatenation to occur. When
;; no such token has been examined immediately return the
;; (stringy) token recieved and indicate it must have been a
;; stringy token by chaning the value of `w--first-stringy-token?'
;; accordingly. Subsequent runs will then operate on potential
;; pairs of stringy tokens.
(if-let ((not-first-stringy-token? (not w--first-stringy-token?))
(previous-token  (pop peg--stack))
;; The previous token cannot be a text or nwnl token if
;; it is not a list, and checking prevents causing an
;; error by taking the `car' of a non-list token, e.g. the
;; filename token.
(previous-token-is-alist?
(prog1 (and (listp previous-token)
(listp new-token)
(or (assoc 'text `(,new-token))
(assoc 'nl `(,new-token)))
(or (assoc 'text `(,previous-token))
(assoc 'nl `(,previous-token)))))))
;; Join the association's values and let the caller push a single
;; token back onto the `peg--stack'.
(cons 'text (format "%s%s" (cdr previous-token)
(cdr new-token)))

;; Push the previous token back to the `peg--stack', and let the
;; caller push the new token to that stack.
(push previous-token peg--stack)
new-token)
(when w--first-stringy-token?
(setq w--first-stringy-token? nil))))
```

Defines:
`w--concatenate-text-tokens`, used in 10 chunk 0.

19 ⟨push the compiled SQL to the database and to the history stack 19⟩≡
```
;; NOTE: the result of evaluating the SQL is pushed to the history stack
;; alongside the SQL that was executed.
(cl-pushnew (cons (emacsql (w-project-database-connection default-project)
compiled-parse-tree)
. compiled-parse-tree)
(w-project-history-sql-commands default-project))
```

.1 Packaging whyse

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.

19 ⟨whyse.el 19⟩≡
⟨Emacs Lisp package headers 19⟩
⟨Licensing and copyright 20⟩
⟨Commentary 20⟩
⟨Code 18⟩
⟨provide the whyse feature and list the file local variables 21⟩

19 ⟨Emacs Lisp package headers 19⟩≡
```
;;; whyse.el --- noWeb HYpertext System in Emacs -*- lexical-binding: nil -*-
;; Yes, you read that right: no lexical binding in this file.

;; 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 2.

19 ⟨whyse-pkg.el 19⟩≡
```
(define-package "whyse" "0.1" "noWeb HYpertext System in Emacs"
'(⟨required packages 20⟩))
```

Uses `whyse` 1 2.

The following chunk lists the ⟨required packages 20⟩; as of `whyse-0.1-devel` the only required packages are `peg` and `cl-lib`.

20 ⟨required packages 20⟩≡
```
(emacs "25.1")
(emacsql "20230220")
(peg "1.0.1")
(cl-lib "1.0")
```

20 ⟨Licensing and copyright 20⟩≡
```
;; 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/>.
```

20 ⟨Commentary 20⟩≡
```
;;; 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).
```

20 ⟨Code 18⟩+≡
```
;;; Code:
;;;; Compiler directives
(eval-when-compile (require 'wid-edit))

;;;; Internals
⟨Customization and global variables 1⟩
⟨Widgets 1⟩
⟨WHYSE project structure 3⟩
⟨with-project 14⟩
⟨buffer parsing function 15⟩
⟨open-project-hook 5⟩
⟨default hook functions 5⟩

;;;; Commands
;;;###autoload
```

⟨WHYSE 2⟩

21 ⟨provide the whyse feature and list the file local variables 21⟩≡

```
(provide 'whyse)

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

.1.1 TESTING

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.

TODO Adopt/use `makem.sh`, by "alphapapa".

Parsing tool syntax within a temporary buffer

22 ⟨test-parser-with-temporary-buffer.el 22⟩≡

```
;; -*- lexical-binding: nil; -*-
(defvar w--parse-success t
"A simple boolean regarding the success or fialure of the last
attempt to parse a buffer of Noweb tool syntax.")

⟨buffer parsing function 15⟩
(with-temp-buffer
(insert (shell-command-to-string
"make --silent --file ~/src/whyse/Makefile tool-syntax"))
(goto-char (point-min))
(w--parse-current-buffer-with-rules))

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

Defines:
`w--parse-success`, used in 15 chunk 0.
Uses `w--parse-current-buffer-with-rules` 15 and `whyse` 1 2.

.2 Indices

.2.1 Chunks

⟨fundamental indexing keywords, which are restricted to within a code chunk 12⟩
⟨high-level Noweb tool syntax structure 7⟩
⟨identifiers defined in a chunk 12⟩
⟨identifiers used in a chunk 12⟩
⟨index nl error message 14⟩
⟨indexing and cross-referencing set-off words 12⟩
⟨keyword definitions 11⟩
⟨Licensing and copyright 20⟩
⟨map over SQL s-expressions, creating the tables 5⟩
⟨meta rules 7⟩
⟨open-project-hook 5⟩
⟨PEG rules 7⟩
⟨provide the whyse feature and list the file local variables 21⟩
⟨push the compiled SQL to the database and to the history stack 19⟩
⟨quotations 11⟩
⟨required packages 20⟩
⟨return a filename for the project database 4⟩
⟨setup project database 4⟩
⟨structural keywords 9⟩
⟨structural keywords (except quotations) 9⟩
⟨tagging keywords 10⟩
⟨test-parser-with-temporary-buffer.el 22⟩
⟨the index of identifiers 12⟩
⟨tool errors 10⟩
⟨unsupported indexing keywords 13⟩
⟨WHYSE 2⟩
⟨WHYSE project structure 3⟩
⟨whyse-pkg.el 19⟩
⟨whyse.el 19⟩
⟨Widgets 1⟩
⟨with-project 14⟩

.2.2 Identifiers

Underlined indices denote definitions; regular indices denote uses.

.3 NOTES

List of notes