Coda: Code Documentation Application

Agnar Renolen

2.0, July 2001

Contents

1 Introduction

WHAT IS CODA?

A coda is a passage that completes a piece of music. Likewise, the Coda code documentation application is a tool that completes the development of software: When the final version has been implemented, before release, it is time to produce the documentation.

Coda reads specially formatted comments in the source code, and produces output in various formats. This version of Coda is planned to support output in HTML and LaTeX. Coda can also read comments from several programming languages. Coda is packagesd with support for Tcl, Perl, C, C++, C#, Java, Pascal, Lisp and Visual Basic. However Coda can be customized to support other formats as well. Coda also support reading pure text files containing raw CodaText.

RUNNING CODA

The calling syntax of coda is as follows:

`coda [-q] [-t` *format*`] [-d` *dir*`] [-r` *pattern*`] [-f` *file*`]` *pattern1* `[`*pattern2* `...]`

The parameters have the following meaning.

`-t` Specifies the output format. Leagal values of *format* are "html" and "latex". The default value is "html".

`-d` Specifies the target directory where coda is to put the output files. The default value is "doc"

`-r` Specifies that all files in the current directory and in all subdirectories matching a *pattern* is to be docified. This options can be repeated many times with different patterns.

`-f` Specifies a file named *file* contains a list of files to docify. This option is also repeatable.

`-q` Tells coda to be quiet, that is, no output will be given during progress.

The remaining arguments, *pattern1*, *pattern2* and so forth, specifies the files to docify in the current directory.

THE CODA FORMAT

The text format regognized by coda is called *CodaText*. It is a text format designed for documenting source code by writing the documentation as comments within the source files. The main purpose of designing this format is to provide a text format that is

- easy to read
- easy to write
- easy to remember
- fairly easy to parse

You may read the CodaText format specification[1] about the CodaText format.

2 coda

2.1 coda::BeQuiet

SYNOPSIS

`BeQuiet`

DESCRIPTION

Makes coda run in quiet mode, that is, no output will be produced when coda is running. In practice, since all output is performed using the `puts` function, the proc renames the original `puts` proc to `fputs` and creates a new `puts` function that forwards it's procs to `fputs` only if output is to a channel.

2.2 coda::DocifyFile

SYNOPSIS

`DocifyFile` *fileName*

[1] http://coda.sourceforge.net/codatext.html

DESCRIPTION

This proc checks the extension of the *fileName* and deploys the corresponding reader. If the file type is not regognized, the proc outputs a message, and returns without docifying the file.

2.3 coda::DoDocify

SYNOPSIS

`DoDocify ?`*options*`? ?`*pattern*`? ?`*pattern*`? ...`

DESCRIPTION

Docifes a project. Regognized options are:

`-t` *format* Specifies a target format. Legal values are `html` wich is default, and `latex`.

`-d` *dirName* Specifies a target directory. Default value is `doc`.

`-r` *pattern* Specifies that all files matching *pattern* in the top directory and all subdirectories are to be docified. This option is repeatable.

`-f` *fileName* Specifies that the file *filename* contains a list of files to be docified. This options is repeatable.

The remaining *pattern* arguments specifies that all files mathing the *pattern* in the current directory are to be docified.

2.4 coda::DoListFile

SYNOPSIS

`DoListFile` *fileName*

DESCRIPTION

The file *fileName* contains a list of files, and this proc will docify all of them.

2.5 coda::DoRGlob

SYNOPSIS

`DoRGlob` *pattern*

DESCRIPTION

Implements the `-r` *pattern* options of Coda. Docifies all files matching a *pattern* in the top directory (either specified in the pattern or in the current directory) and all its subdirectories.

2.6 coda::GetOutputFormats

SYNOPSIS

`GetOutputFormats`

DESCRIPTION

Returns a list of output formats supportede by coda.

2.7 coda::Glob

SYNOPSIS

`DoGlob` *pattern*

DESCRIPTION

Docifies all files mathing *pattern* in the current or specified directory. In contrast to coda::DoRGlob (see Section **??**), this proc does not search through subdirectories.

2.8 coda::MakeFileName

SYNOPSIS

`MakeFileName` *fileName*

DESCRIPTION

If a doc item contains colons, for example because of namespaces, then returns a string which is identical to *name* except that all colons, semicolons and amphersand's are replaced with underscores (Remember filenames cannot contain colons these characers).

2.9 coda::SetOutputFormat

SYNOPSIS

`SetOutputFormat` *format*

DESCRIPTION

Specifies the output format that coda shuld produce. If the format is not supported by coda, an error is generated. Standard formats supported by coda is currently "html" and "latex".

3 CodaText

DESCRIPTION

CodaText is a text format that is designed to be easy to read and write as source text, and fairly easy to parse and produce output in for example html. The basic idea is to minimize the markup and make it into a natural part of the text. You can read the CodaText format specification[2] for the details of the format.

This namespace provides the basic routines to parse up text in CodaText format. The only proc that you should need to call is CodaText::Parse (see Section **??**).

3.1 CodaText::BeginEnvironment

SYNOPSIS

`BeginEnvironment` *environmentType indentation*

DESCRIPTION

Begins a new environment. This proc is called by CodaText::BeginParagraph (see Section **??**) whenever it starts a paragraph that needs a new environment. The environment types reckognized by CodaText is itemized lists (il), enumerated lists (el) and description lists (dl).

3.2 CodaText::BeginParagraph

SYNOPSIS

`BeginParagraph` *paragraphType indentation*

DESCRIPTION

Called whenever CodaText::ParseBody (see Section **??**) encounters a new paragraph. This proc checks the *paragraphType*, which is one of those returned by FindLineType (see Section **??**), and sets up the correct environemnt. If it needs to be changed, calls CodaText::BeginEnvironment (see Section **??**) and CodaText::EndEnvironment.

3.3 CodaText::DoOutput

SYNOPSIS

`DoOutput` *key value*

[2] http://coda.sourceforge.net/codatext.html

DESCRIPTION

Appends a new *key* - *value* pair to the output list, which eventually will be returned by CodaText::Parse (see Section **??**). This proc also keeps tracks of the stacks, and gives a warning if tags are not problerly formatted.

3.4 CodaText::EndEnvironment

SYNOPSIS

`EndEnvironment`

DESCRIPTION

Ends the current environment. This proc is called by CodaTExt::BeginParagraph (see Section **??**) whenever it starts a paragraph that requires the current environment to end.

3.5 CodaText::EndParagraph

SYNOPSIS

`EndParagraph` *pTypeVar pTextVar indentVar*

DESCRIPTION

Called whenever CodaText::ParseBody (see Section **??**) encounters the end of a paragraph. *pTypeVar* is the name of a variable specifying the type of paragraph to be ended, *pTextVar* is the name of a variable holding the raw paragraph text, and *indentVar* is the name of a variable holding the indentation if the paragraph.

The proc calls CodaText::ParseParagraph (see Section **??**) to parse up the style markup of the paragraph, and then set the paragraph type variable and the paragraph text variabl to "", and the indentation variable to "0". The reason for clearing the variables in this proc, is to save the labour of doing it after each call to this proc.

3.6 CodaText::FindLineType

SYNOPSIS

`FindLineType` *string textVar indentVar paramVar*

DESCRIPTION

Checks *string* whether it contains any indication of a new paragraph. Returns

h1 If *string* starts a level1 heading

h2 If *string* starts a level2 heading

ii If *string* starts a bulleted item

ei If *string* starts a enumerated item

pre If *string* starts a preformatted section

dt If *string* is a keyword of a description item

hr If *string* is a horizontal rule

"" If *string* is empty.

fig If *string* describes a figure or image. The image filename will be stored in the *paramVar* variable.

p Otherwise.

The variable named *textVar* will contain the text of the line, when the paragraph formatting tags has been removed. The variable named *indentVar* will contain the number of spaces in the beginning of *string*. If the paragraph marker contains an argument, such as
`fig`{*filename*}, the value of the argument will be stored in the variable named *paramVar*.

CHANGES

12.05.2002 Extended proc to be able to return argument values of certain paragraph markers.

3.7 CodaText::Parse

SYNOPSIS

`Parse` *headerVar textLines*

DESCRIPTION

Parses the list *textLines*, which contains the text lines of the input, stripped from foreign markup (such as comment tags). *headerVar* is the name of an array into which the Parser will but the header information it encounters. The array will at least contain the following entries:

`name` Contains name of the doc item.

`type` Specifies the type of doc-item. Defaults to "item" if omitted. If type is set to "document", will normally generate a standalone output file.

Although the header will contain all entries from the header of the doc-item, the following entries are considered standard:

`title` For larger documents, a title is appropriate. If you are documenting a procedure, it is recommended to leave the title undefined.

`summary` Gives a brief one-line summary of the doc-item

`author` Contains a list of all authors

`version` Specifies the version of the document.

`date` Specifies the date of writing, or the date of approval of the document.

`mansection` If output is in unix man page format (roff), specifies the section into which the man page is to be put.

The proc returns a list on the following format (which is similar to the `text dump` tk command):

key1 value1 arg1 key2 value2 arg2 ...

The contents of *value* depends on the value of *key*. The possible keys are as follows:

`begin` begins a new environment, paragraph or style. The *value* is the name of the tag. Possible environment tags are:

`il` An itemized list

`el` An enumeretated list

`dl` A description list

Possible paragraph tags are:

`p` An ordinary paragraph.

`dt` The description term in a description list.

`pre` Preformatted text.

`fig` A figure. A `begin fig` pair will always be succeeded by a `file` key .

Possible styles are:

`em` Emphasized text

`code` A snippet of code, input or output.

`var` A metasymbol, typically a variable.

`ref` A reference to another doc-item

`link` A hyperlink to an url. This key will allways be succeded by an `url` key.

`cmd` A command, typically a menu command.

`end` Ends the current environment, paragraph or style. The *value* is the name of the tag, and can have the same values as for the `begin` key.

`item` Starts an item of itemized or enumerated list. For itemized lists, the *value* is "*", for enumerated lists, the *value* will be the number of the item. Note that text within these lists will have the normal paragraph tag. There can be several paragraphs under the same item.

`url` This key will always follow a `begin link` pair, and the *value* will be the url that the link points to.

`text` Specifies that *value* will contain a block of text.

3.8 CodaText::ParseBody

SYNOPSIS

`ParseBody` *bodyLines*

DESCRIPTION

Parses the body of a doc-item. The *bodyLines* argument is a lst of text lines comprising the body.

CHANGES

12.05.2002 Proc extended to handle `fig` key.

3.9 CodaText::ParseHeader

SYNOPSIS

`ParseHeader` *headerVar headerLines*

DESCRIPTION

Parses the header of a CodaText item that adheres to the standard header formet. The *headerVar* is the name of an array that will contain the header information, and *headerLines* is a list of text lines comprising the header of a CodaText document.

See CodaText::Parse (see Section **??**) for the contents of the header array.

3.10 CodaText::ParseParagraph

SYNOPSIS

`ParseParagraph` *paragraphType paragraphText*

DESCRIPTION

Parses *paragraphText* for markup in running text. The following regognized markup tags are recognized:

emphasized text Is identified as the text between two forward /slahses/.

computer voice Is identified as the text between square [brackets].

metasymbols Is identifies as the text between ¡angular braces¿.

references A reference to another document item is identfied as the text following a dollar sign and ended by the first space or dollarsign.

hyperlinks A link to an url is identified as the text embraced in curly braces prefixed by the at charcter.

commands Commands, such as menu commands are embraced between two —vertical bars—

3.11 CodaText::ParseShortHeader

SYNOPSIS

`ParseShortHeader` *headerVar headerLines*

DESCRIPTION

Parses the header of a CodaText item that adheres to the short header format The *headerVar* is the name of an array that will contain the header information, and *headerLines* is a list of text lines comprising the header of a CodaText document.

See CodaText::Parse (see Section **??**) for the contents of the header array.

4 EmbraceReader

DESCRIPITON

The Embrace reader is a comment reader that extraxts coda-comments from programming languages where comments are embraced in special characters such as "/*" and "*/" for C and java, and "{" and "}" for Pascal.

For the EmbraceReader to recognize coda-comments it needs a *coda prefix* which distinguishes comments written for coda and other comments. The coda prefix must be the only characters on the line, and the Coda comment applies to the whole comment started by the coda prefix, and ended by the *comment suffix*.

In addition, the EmbraceReader alows each line in the comment block to be prefixed with a character to make the comment stand out more from the source code, as in the follwing example.

```
/**
 * EmbraceReader --- Comment reader for embrace style comment syntax
 *
 * DESCRIPTION
 *   ...
 */
```

In order for the EmbraceReader to trim off this *line prefix* correctly, it must be alligned with the first identical character in the coda prefix, as in the example above. This rule is necessary in order to distinguish the line prefix from the bullet of an itemized list.

AUTHOR

Agnar Renolen, July 2001.

4.1 EmbraceReader::GetNextItem

SYNOPSIS

`GetNextItem` *channelID*

DESCRIPTION

Reads the next comment from the source file read in *channelID*. If a coda comment block is read, the contents of the comment is returned (stripped from comment markup) in form of a list of text lines. If no comment is found, an empty string is returned, indicating that the end of the file has been reached.x

CHANGES

Jan 2nd, 2001 Previous version did not work for comments which did not include a line prefix as indentation was lost. This un-reported bug is now fixed.

4.2 EmbraceReader::Init

SYNOPSIS

`Init` *codaPrefix linePrefix commentSufix*

DESCRIPTION

Sets up the EmbraceReader to recognize comments where the first line starts by *codaPrefix*, and each line can be prefixed with a *linePrefix* and ended by a *commentSuffix*. For example for Java, one should set the coda prefix to "/**", the line prefix to "*" and the comment suffix to "*/". Note that the *linePrefix* must be contained in the comment suffix, in order to be regognized.

5 HtmlOut

DESCRIPTION

The HtmlOut namespace provides procs for producing output of parsed Coda-Text in HTML. The main procs of the namespace are

- HtmlOut::BeginProject (see Section **??**)
- HtmlOut::Docify (see Section **??**)
- HtmlOut::EndProject (see Section **??**)

This docifier will output the files into a directory known as the *target directory*. Each doc-item will be given it's own output file. And when the project is ended, it copies the `coda.css` file from the home directory to the target directory. Then produces a main file named `index.html`, and a contents file named `toc.html`. The main file, will set up three frames in the browser. The left frame will contain the `toc.html` file and the right frame will contain an upper frame for the title and a lower main frame for the doc-item.

If a doc-item is named `main`, it will be set up as the initial file in the main frame. Otherwise, the first item in the table of contents frame, which should be sorted alphabetically, will be set up as the intial file in the right frame.

Doc-items that has the `type` entry of the header set to "document", will not be included in the table of contents. Use this if you want to create stand-alone html pages that is not included included in the large set of files.

5.1 HtmlOut::BeginProject

SYNOPSIS

`HtmlOut::BeginProject ?`*options*`?`

DESCRIPTION

Begins a new docify project, and creates a new directory for the output if necessary. Valid options are:

`-dir` Specifies the output directory. The dafault value is "doc".

5.2 HtmlOut::Docify

SYNOPSIS

`HtmlOut::Docify` *headerVar docData*

DESCRIPTION

Docifies a doc-item. Generates a file named after the name of the doc-item. The *headerVar* array and the *docData* list contains data returned by CodaText::Parse (see Section **??**).

This proc calls in turn:

1. HtmlOut::RegisterName (see Section **??**) to register the name of the doc-item for the table of contents file for the project.
2. HtmlOut::PutHeader (see Section **??**) to generate the html header and the header of the doc item
3. HtmlOut::PutTOC (see Section **??**) to generate a table of contens entry of the doc-item.
4. HtmlOut::PutBody (see Section **??**) to generate the body text of the

doc-item.

5.3 HtmlOut::DoTags

SYNOPSIS

`DoTags` *tagList type*

DESCRIPTION

If *type* equals "start", creates a string that starts the html tags in the *tagList*. For example, the command `DoTags {p b} start` would return the string "*pb*". If *type* equals "end" produces the corresponding end tasks, in reverse order. For example, the command `DoTags {p b} end` will return the string "b*p*".

5.4 HtmlOut::EndProject

SYNOPSIS

```
EndProject
```

DESCRIPTION

Called when all doc-items has been docified. This proc creates the main article which will include all other items in alphabetical order.

5.5 HtmlOut::EndProject

SYNOPSIS

```
EndProject
```

DESCRIPTION

Called when all doc-items has been docified. This proc creates the main article which will include all other items in alphabetical order.

5.6 HtmlOut::MakeReferenceText

SYNOPSIS

`MakeReferenceText` *refName*

DESCRIPTION

If a doc item named `HtmlOut::PutBody` has a reference to `HtmlOut::MakeReferenceText` then the output for PutBody should not include the full name "HtmlOut::MakeReferenceText", but rather just "MakeReferenceText". This proc returns "MakeReferenceText" if the the text is in the same namespace as the current item, or returns the complete name otherwise.

5.7 HtmlOut::PutBody

SYNOPSIS

`PutBody` *channelID docData docType*

DESCRIPTION

Writes the body of a doc-item to the file inentified by *channelID*. The *docData* is a list returned by CodaText::Parse (see Section **??**), and *docType* is the value of the `type` entry of the header array.

5.8 HtmlOut::PutFigure

SYNOPSIS

`PutURL` *channelID URLorFile*

DESCRIPTION

Invoked by HtmlOut::PutBody (see Section **??**) whenever an file key is encountered.

5.9 HtmlOut::PutHeader

SYNOPSIS

`PutHeader` *channelID headerVar*

DESCRIPTION

Generates the header of a doc-item and writes out to *channelId*. Two types of headers are generated, depending on the contents of the array named *headerVar*. If the *headerVar* array contains an entry named `title`, generates an article-style header, otherwise generates an abbreviated header styles suitable for documenting procs like this.

5.10 HtmlOut::PutTOC

SYNOPSIS

`PutTOC` *channelID docData*

DESCRIPTION

Traces through *docData* which is the parsed codaText returned by CodaText::Parse (see Section **??**) and generates a table of contents for the current doc item. The contents are written out to the *channelID* by extracting thos paragrahs that are of type `h1` and `h2`, under a level one header "CONTENTS"

5.11 HtmlOut::RegisterName

SYNOPSIS

`RegisterName` *name*

DESCRIPTION

This proc registers a *name* for the content index in the left frame of the browser. The proc will attempt to register the name in a hierarchical fashion. The names are organized hieracally be separating levels by period like in *myclass*.*method* or by colons like in *myclass*::*method*.

5.12 HtmlOut::WriteMainFile

SYNOPSIS

`WriteMainFile` *firstItem*

DESCRIPTION

Sets up the main html file named `index.html`. The sets up two frames in the browser, where the left frame contains the file generated by HtmlOut::WriteTocIndex (see Section **??**) and the right frame contains the file containing the item specified by the *firstItem* argument.

5.13 HtmlOut::WriteStyle

SYNOPSIS

`WriteStyle`

DESCRIPTION

Copies a style file from the coda directory into the target directory.

5.14 HtmlOut::WriteTocIndex

SYNOPSIS

`HtmlOut::WriteTocIndex`

DESCRIPTION

Writes out the html file which will appear in the left frame of the browser containing an alphabetical and hiearcical list of all doc-items of the project. The proc calls HtmlOut::WriteTocItems (see Section **??**) to create the index itself by a recursice method. This proc sets up the file, and the other stuf for the index.

5.15 HtmlOut::WriteTocItems

SYNOPSIS

`HtmlOut::WriteTocItems` *channelID*

DESCRIPTION

Writes out the the table of contents to the file identified by *channelID*.

CHANGES

`Jan 2nd, 2002. by Agnar Renolen` Changed from a recursive procedure to a straight one due to change in the way the *tableOfContents* variable is stored.

6 LatexOut

DESCRIPTION

LatexOut is a docifier that produces output in LaTeX format. It works as follows: Each doc item that have the type set to "document", will generate its own article, otherwise it will be included in the *main article*. If an item is named "main", it will be the first item to appear in the main article. The other items will be reproduced in alphabetical order. The main item may contain a

title, which will be the title of the main article. If there is no main item or the main item does not have a title, the title of the main article will be "Coda Documentation".

6.1 LatexOut::BeginProject

SYNOPSIS

`BeginProject ?`*options*`?`

DESCRIPTION

Begins a new project, and creates a new directory for the output if necessary. Valid options are:

`-dir` Specifies the output directory. The dafault value is "doc".

6.2 LatexOut::CheckString

SYNOPSIS

`LatexOut::CheckString` *string*

DESCRIPTION

Checks *string* for characters that are special to LaTeX and returns a string where these characters are escaped with a backslash.

6.3 LatexOut::Docify

SYNOPSIS

`HtmlOut::Docify` *headerVar docData*

DESCRIPTION

Docifies a single doc-item. If the doc-item is of type "document", generates a stand-alone article for that item. Otherwise, generates the body text of a section of the main article. This article will be included in the main article generated by LatexOut::EndProject (see Section **??**). The proc calls LatexOut::PutBody (see Section **??**) to generate the body of the item.

6.4 LatexOut::PutArticleHeader

SYNOPSIS

`PutArticleHeader` *channelID headerVar*

DESCRIPTION

Generates the header of a LaTeX document of class "article". The output is written to *channelID* and the *headerVar* is the name of an array containing the header from the coda parser.

6.5 LatexOut::PutBody

SYNOPSIS

`PutBody` *channelID headerVar docData*

DESCRIPTION

Formats the body of a doc-item in LaTeX format. The output is written to *channelID*, and the *headerVar* is the name of the array of the header info returned from the coda parser, and the *docData* is the parsed data generated by the parser.

6.6 LatexOut::PutItemHeader

SYNOPSIS

`PutItemHeader` *channelID headerVar*

DESCRIPTION

Writes out the header of each doc item. Specifically, this proc checks the name, and inserts a new section if the toplevel namespace is new.

Then it adds a new subsection for the doc item; using the title if it exists, the name otherwise.

6.7 LatexOut::PutParagraph

SYNOPSIS

`PutParagraph` *channelID paragraphText*

DESCRIPTION

This proc will output the *paragraphText* containing formatted LaTeX to *channelID*. Since *paragraphText* might be longer than one line long, it splits it into several lines in order to make the output readable in ordinary editors.

7 misc

DESCRIPTION

This name space is imported into the root namespace. It is therefore not necessary ti call them by the prefix.

7.1 misc::FindCommonStart

SYNOPSIS

`FindCommonStart` *list*

DESCRIPTION

Searches through the *list* and returns the longest substring that all items in the list begins-with

7.2 misc::html

SYNOPSIS

`html` *tag string* `?`*-param value -param value* `...?`

DESCRIPTION

Returns a string where *string* is emebedded between an opening and closing html-tag of type *tag*. For example the tcl command

```
html h2 "This is a Header"
```

Would return the string `<h2>This is a Header</h2>`.

The optional *-param value* arguments provide a convenient way to provide parameter arguments to the tag. For example the tcl command

```
html h2 "Centered Header" -align center
```

Would return the string `<h2 align="center">Centered Header<h2>`

Three values of *string* have a special meaning:

`+` Creates only an opening tag

`-` Creates only a closing tag

`-+` Creates a closing tag followed by a new opening tag

7.3 misc::mset

SYNOPSIS

`mset` *varlist valuelist*

DESCRIPTION

Sets multiple variables with values from a list. The *varlist* contains variable names and each variable will be assigned the corresponding value in *valuelist*. For example

```
mset {name age sex} {Bill 42 male}
```

It is an error if *valuelist* contains fewer elements than the *varlist*. If the valuelist contains more elements than the *varlist* the exceeding values will not be assigned to any variable.

The procedure returns the a list containing the elements in *valuelist* that were no assigned to a variable.

7.4 misc::PlaceDialog

SYNOPSIS

`PlaceDialog` *dialogWindow* `?`*parentWindow*`?`

DESCRIPTION

Places a toplevel *dialogWindow* horizontally centered, and centered vertically one third from the top on a *parentWindow*. If *parentWindow* is omitted, then it is places according to the desktop.

7.5 misc::rglob

SYNOPSIS

`rglob` *pattern*

DESCRIPTION

Executes a recursive *glob* returning a list of all files mathing a *pattern*. If the search is to be started in a particular directory, this is obtained by prefixing the *pattern* with the directory name as e.g., `rglob "c:/develop/tclproject/*.tcl"`.

The globbing is executed using the `-nocomplain` option, so if `rglob` finds no matches, an empty list is returned.

7.6 misc::setoptions

SYNOPSIS

`setoptions -`*option1* `?`*value1*`?` `-`*option2* `?`*value2*`?` `...` `?--?` `?arg1 arg1 ...?`

DESCRIPTIONS

Provides a convenient way to parse options passed as parameters to procedures. The procdure will create variables named *option*i which will be assigned to the corresponding value of *value*i. If no value is provided for an option, the variable will be assigned the value `true`. A `--` marks the end of options. You must use this option if the last option is not to be assigned any particular value (other than true) to prevent the procedure to assign the value of the first *arg* to the the last option.

The method returns a list consisting the remaining arguments *arg1 arg2 ....* For example, the command

`setoptions {-name Bill -age 42 -sex male -developer -- hello world}`

Will set the variable `name` to "Bill", `age` to "42" `sex` to "male" and `developer` to true, and return the list {`hello world`}.

7.7 misc::stack

SYNOPSIS

`stack` *option varName* `?`*arg*`?` `?`*arg ...*`?`

DESCRIPTION

Performs one of several stack operations on a stack given by *varName*. The legal *option*s are:

`stack depth` *varName* Returns the number of elements in the stack

`stack pop` *varName* Pops the topmost element from the stack and returns the popped element.

`stack push` *varName* `?`*value value value*`...?` Pushes one or more elements on the stack. The last *value* will be the topmost element in the stack.

`stack top` *varName* Returns the topmost element on the stack without popping it from the stack.

A stack is simlpy a list where the topmost element is at the end of the list.

8 PlainReader

DESCRIPTION

The `PlainReader` is a reader that reads files that are written in pure CodaText. One doc-item per file. This reader also allows the author to add comments in the file. A comment is recognized as a line with a percent ("%") as the first (non-space) character.

8.1 PlainReader::GetNextItem

SYNOPSIS

`GetNextItem` *channelID*

DESCRIPTION

Reads the entire input from *channelID* and return the contents in the form of a list of text lines. Lines that are comments, which are identified by a percent ("%") as the first non-space character, are added to the list as empty lines. This is to make sure that the parser reports the correct line number of an syntax error is encountered.

If the end of the file has been encountered, returns an empty string.

8.2 PlainReader::Init

SYNOPSIS

`Init`

DESCRIPTION

This proc does nothing, but is provided to make it comaptible with other readers.

9 PrefixReader

The prefix reader is a comment reader that extracts doc-comments from programming languages where comments are prefixed with a special character or set of characters. In tcl this is the pound character (#), while in Lisp, it is the semicolon character. In order to distinguish Coda-comments from other comments, a Coda comment block must begin with a superstring of the initial character. For tcl, like in this source file, Coda-comments starts with a double pound.

And additional requirement also applies to comments read by this reader: The prefix must be the first non-space character on the string. The PrefixReader needs two parameters to read any source code that uses a prefix commenting style, like tcl. The prefix itself, and the prefix that starts a coda-comment (so non-coda comments can be discarded).

A block of coda comment is thus identified by the coda comment prefix, and subsequent lines starting with the language comment prefix. In other words, a coda comment ends by the first line that is not a comment.

AUTHOR

Agnar Renolen, July 2001.

9.1 PrefixReader::GetNextItem

SYNOPSIS

`GetNextItem` *channelid*

DESCRIPTION

Reads the next coda comment from the source file read in *channelid*. If a coda comment block is read, the contents of the comment (stripped from the comment prefixes) is returned. If no comment is found, and empty string is returned, which indicates that the end of file has been reached.

9.2 PrefixReader::Init

SYNOPSIS

`PrefixReader::Init` *codaPrefix commentPrefix*

DESCRIPTION

Sets up the PrefixReader to recognize comments where the first comment line is prefixed with the *codaPrefix*, and the subsequent lines are prefixed with the *commentPrefix*. The *codaPrefix* is necessary in order to distinguish coda comments from any other comments in the source code, and the first characters of the *codaPrefix* must be equal to the *commentPrefix*.

If this proc is not called, the *codaPrefix* will be set to `##` and the *commentPrefix* will be set to `#`.

10 wcoda

SYNOPSIS

`wcoda::show`

DESCRIPTION

Shows a window where the user can select a set of files which the tcldoc program might read comments comments and generate html documentation files into a taregt directory.

10.1 wcoda::AddFiles

SYNOPSIS

`AddFiles`

DESCRIPTION

Invoked when the user adds a file from the current directory to the list of selected files.

10.2 wcoda::BuildStatusWindow

SYNOPSIS

`BuildStatusWindow` *pathName* `?`*parent*`?`

DESCRIPTION

Builds a window into which progress standard output is directed during docification. The window will have a close button with the path name *pathName*`.close`.

10.3 wcoda::BuildWindow

SYNOPSIS

`BuildWindow` *pathName*

DESRIPTION

Builds the main window used by wcoda.

10.4 wcoda::ChangeCurrentDir

SYNOPSIS

`ChangeCurrentDir`

DESCRIPTION

Invoked when the user presses the "Change Dir..." button. Brings up the choose directory button, and filles the current directory listbox with the files of the new directory if one was chosen.

10.5 wcoda::ChangeDir

SYNOPSIS

`ChangeDir` *yCoord*

DESCRIPTION

Invoked when user double clicks an item in the current directory list box. The *yCoord* parameter specifies the y-coordinate of the mouse click. If the click was in a directory, changes to that directory, and calls wcoda::FillWorkingDir (see Section **??**) to fill the list box with files of the new dorectory.

10.6 wcoda::CheckDocability

SYNOPSIS

`CheckDocability`

DESCRIPTION

Checks whether there are files in the listbox containing files to docify, and enables the "Docify" button if it contains files.

10.7 wcoda::CloseStatusWindow

SYNOPSIS

`CloseStatusWindow` *pathName*

DESCRIPTION

Closes the status window, normally invokes when the user closes the close button.

10.8 wcoda::DeleteFiles

SYNOPSIS

`DeleteFiles`

DESCRIPTION

Invoked when the user wants to delete files from the list of selected files.

10.9 wcoda::Docify

SYNOPSIS

`Docify`

DESCRIPTION

Invoked when the user presses the "Docify" button. Brings up the staus window and redirects all output to this window.

10.10 wcoda::FillFormatCombo

SYNOPSIS

`FillFormatCombo` *formatList*

DESCRIPTION

Fills the combobox specifying output format with values from *valueList*.

10.11 wcoda::FillWorkingDir

SYNOPSIS

```
FillWorkingDir
```

DESCRIPTION

Fills the current directory listbox with all files in the current directory.

10.12 wcoda::PickTargetDir

SYNOPSIS

```
PickTargetDir
```

DESCRIPTION

Invoked when the user presses the "browse..." button of the target directory entry. Sets the target directory entry if a directory was chosen in the dialog that appears.