Logiciel

Noweb : documentation automatique de programme (traduction du man)

Voir les traductions de :

- A One-Page Guide to Using noweb with LATEX : Utiliser NoWeb avec LATEX : guide d'utilisation en une page
- The noweb Hacker’s Guide : Le Guide du Hacker Noweb (The noweb Hacker’s Guide)

Man Noweb

literate programing NOWEB(1) General Commands Manual NOWEB(1)

NAME

```
     noweb - a simple literate-programming tool
```

SYNOPSIS

```
     noweb [-t] [-o] [-Lformat] [-markup parser] [file] ...
```

DESCRIPTION

```
     Noweb is a literate-programming tool like FunnelWEB or nuweb, only sim-
     pler.  A noweb file contains program source code interleaved with docu-
     mentation.  When noweb is invoked, it writes the program source code to
     the output files mentioned in the noweb file, and it writes a TeX  file
     for typeset documentation.
```

```
     The  noweb(1) command is for people who don't like reading man pages or
     who are switching from nuweb.  To get the most out of noweb, use notan-
     gle(1) and noweave(1) instead.
```

FORMAT OF NOWEB FILES

```
     A noweb file is a sequence of chunks, which may appear in any order.  A
     chunk may contain code or documentation.  Documentation  chunks  begin
     with a line that starts with an at sign (@) followed by a space or new-
     line.  They have no names.  Code chunks begin with
     <<chunk name>>=
     on a line by itself.  The double left angle bracket (<<) must be in the
     first column.  Chunks are terminated by the beginning of another chunk,
     or by end of file.  If the first line in the file  does  not  mark  the
     beginning  of a chunk, it is assumed to be the first line of a documen-
```

```
tation chunk.
```

```
Documentation chunks contain text that is copied verbatim to the TeX file (except for quoted code). noweb works with LaTeX; the first documentation chunk must contain a LaTeX \documentclass command, it must contain \usepackage{noweb} in the preamble, and finally it must also contain a LaTeX \begin{document} command.
```

```
Code chunks contain program source code and references to other code chunks. Several code chunks may have the same name; noweb concatenates their definitions to produce a single chunk, just as other literate-programming tools do. noweb looks for chunks that are defined but not used in the source file. If the name of such a chunk contains no spaces, the chunk is an ``output file;'' noweb expands it and writes the result onto the file of the same name. A code-chunk definition is like a macro definition; it contains references to other chunks, which are themselves expanded, and so on. noweb's output is readable; it preserves the indentation of expanded chunks with respect to the chunks in which they appear.
```

```
If a star (*) is appended to the name of an output file, noweb includes line-number information as specified by the -Lformat option (or for C if no -Lformat option is given). The name itself may not contain shell metacharacters.
```

```
Code may be quoted within documentation chunks by placing double square brackets ([[...]]) around it. These double square brackets are used to give the code special typographic treatment in the TeX file. If quoted code ends with three or more square brackets, noweb chooses the rightmost pair, so that, for example, [[a[i]]] is parsed correctly.
```

```
In code, noweb treats unpaired double left or right angle brackets as literal << and >>. To force any such brackets, even paired brackets or brackets in documentation, to be treated as literal, use a preceding at sign (e.g. @<<).
```

OPTIONS

t Suppress generation of a TeX file.

o Suppress generation of output files.

Lformat Use format to format line-number information for starred output files. (If the option is omitted, a format suitable for C is used.) format is as defined by notangle(1);

markup parser Use parser to parse the input file. Enables use of noweb tools on files in other formats; for example, the numarkup parser understands nuweb(1) format. See nowebfilters(7) for more information. For experts only.

BUGS

```
     Ignoring unused chunks whose names contain spaces sometimes causes
     problems, especially in the case when a chunk has multiple definitions
     and one is misspelled; the misspelled definition will be silently
     ignored.  noroots(1) can be used as a sanity checker to catch this sort
     of mistake.
```

```
     noweb is intended for users who don't want the power or the complexity
     of command-line options.  More sophisticated users should avoid noweb
     and use noweave and notangle instead.  If the design were better, we
     could all use the same commands.
```

```
     noweb requires the new version of awk.  DEC nawk has a bug in that that
     causes problems with braces in TeX output.  GNU gawk is reported to
     work.
```

```
     The default LaTeX pagestyles don't set the width of the boxes contain-
     ing headers and footers.  Since noweb code paragraphs are extra wide,
     this LaTeX bug sometimes results in extra-wide headers and footers.
     The remedy is to redefine the relevant ps@* commands; ps@noweb in
     noweb.sty can be used as an example.
```

SEE ALSO

```
     notangle(1), noweave(1), noroots(1), nountangle(1), nowebstyle(7),
     nowebfilters(7), nuweb2noweb(1)
     Norman Ramsey, Literate programming simplified, IEEE Software
     11(5):97-105, September 1994.
```

VERSION

```
     This man page is from noweb version 2.11b.
```

AUTHOR

```
     Norman Ramsey, Harvard University.  Internet address nr@eecs.har-
     vard.edu.
     Noweb home page at http://www.eecs.harvard.edu/~nr/noweb.
```

```
                                local 3/28/2001                      NOWEB(1)
```

Introduction

Programmation Lettrée en utilisant noweb (Andrew L. Johnson and Brad C. Johnson, December 19, 2000)

Changeons notre méthode traditionnelle de construction de programmes : Au lieu d'indiquer à un ordinateur que faire, focalisons-nous sur le fait d'expliquer à un humain ce que nous voulons que l'ordinateur fasse. (Donald E. Knuth, 1984).

C'est le but essentiel de la programmation lettrée (LP en raccourci)

Un tel environnement inverse la notion d'inclusion de la documentation, sous forme de commentaires, dans le code, en celle où le code est imbriqué dans la description d'un programme.

Ainsi, la programmation lettrée facilite le développement et la présentation de programmes informatiques qui suivent de près le cheminement du problème à la solution.

D'où des programmes plus faciles à déboguer et à maintenir.

En programmation lettrée, on spécifie la description et le code du programme dans un dossier source unique, dans l'ordre le plus compréhensible par un humain.

Le code de programme peut être extrait et rassemblé sous forme intelligible pour le compilateur ou l'interprète par un processus appelé **tangling**.

La documentation est produite par un processus appelé **weaving** qui combine la description et le code sous une forme affichable ou imprimable (le plus souvent par TEX ou le LATEX).

Beaucoup d'outils ont été créés pour la programmation lettrée, la plupart fondés, directement ou conceptuellement, sur le système de WEB créé par D. E. Knuth [cf. 1984. Literate Programming. The Computer Journal (27)2:97-111]. Cet article présente **noweb** de Normand Ramsey - un instrument de programmation lettrée simple à utiliser, extensible et indépendant du langage de programmation.

Aperçu du Système noweb

Pour écrire un programme lettré pour **noweb**, créez un fichier texte simple, traditionnellement d'extension **.nw**) dans lequel vous fournissez toute la documentation technique des différentes parties du programme, avec le code source réel de chaque partie du programme.

Ce fichier (voir Listing 1), que nous appellerons le *fichier source nw*, est alors traité par **noweave** pour créer la documentation sous une forme lisible (la version formatée du programme, voir Figure 1), ou traité par notangle pour extraire les morceaux de code et les rassembler dans leur bon ordre pour le compilateur ou l'interprète (la version exécutable du programme, voir listing 2).

Ces deux processus ne sont pas des programmes simples mais un ensemble de filtres à travers lesquels est filtré le fichier source nw.

C'est ce système de pipe qui rend noweb flexible et extensible car les pipes peuvent être modifiés et de nouveaux filtres peuvent être créés et insérés dans les pipes pour changer le processus de noweb.

Listing 1 : fichier source nw

```
\documentclass[10pt]{article}
\usepackage{noweb}
\noweboptions{smallcode,longchunks}
\begin{document}
\pagestyle{noweb}
@ \paragraph{Introduction}
This is [[autodefs.perl]]\footnote{Copyright 1997, Andrew L.
Johnson and Brad C. Johnson, All rights reserved.},
```

```
a Perl script to be used as an [[autodefs]] filter
in the [[noweb]] pipeline to identify and index
some common Perl definitions. Since this
file is also meant to show off some of the
features of [[noweb]] it is purposely verbose
and contorted.
Perl does not require the formal declaration or typing of
variables which makes it difficult to
differentiate between declarations and usages of
variables. We may however find definitions of [[sub]]'s and
[[package]]'s with little difficulty and that is the purpose of
this module. Before we begin we need to know
some facts about [[noweb]]'s pipeline structure.\footnote{Noweb's
pipeline structure is described in the \textit{Noweb Hackers
Guide} which is included in the [[noweb]] distribution.}
Actual code in the pipeline lie between lines
of the form [[@begin code]] and [[@end code]].
In Perl these are easily recognized by the following regular
expressions.
<<Global variables>>=
$begin_code_pat = "^\@begin code";
$end_code_pat
= "^\@end code";
@ %def $begin_code_pat $end_code_pat
2@ Within a code block there are many types of lines.
that contain actual code are prefixed by [[@text]].
<<Global variables>>=
$code_line_pat = "^\@text";
@ %def $code_line_pat
Ones
@ If, on a code line inside a code block, we find something that
should be added to the ''Defines'' block at the end of the code
chunk and appear in the index, then we need to add a line to the
pipeline of the form ''[[@index defn <ident>]]''.
<<Global variables>>=
$index_prefix = "\@index defn";
@ %def $index_prefix
@ \paragraph{autodefs.perl}
Our actual Perl script has the following simple shape:
<<autodefs.perl>>=
#!/usr/bin/perl
<<Global variables>>
<<[[process_code_chunk]] subroutine>>
while ( <> ) {
    print $_;
    if (/$begin_code_pat/o) {
        process_code_chunk;
    }
}
@
\paragraph{Processing the code chunk}
```

```
To process the code chunk we need to perform a few housekeeping
tasks. First, we only want to consider lines that begin with
[[$code_line_pat]] and second, we want to stop when we find a line
that matches [[$end_code_pat]]. The following loop will suffice
for this purpose.
<<[[process_code_chunk]] subroutine>>=
sub process_code_chunk {
    while ( ($_ = <>) && !/$end_code_pat/o ) {
        print $_;
        if( /$code_line_pat/o ) {
            <<Find and print any definitions>>
        }
    }
    print $_; # make sure we print the ‘‘@end code’’ line
}
@
@ When checking for definitions we first strip off
any comments since [[sub]] or [[package]] may
also occur in a comment. We then build
a list [[@def_list]] which contain all of the
3[[sub]] and [[package]] definitions on the line
and print out an [[@index defn]] line for
each.
<<Find and print any definitions>>=
$_ =~ s/#.*//o;
@def_list = (/sub\s(\w+)/go, /package\s(\w+)/go);
foreach $item (@def_list) {
    print "$index_prefix $item\n";
}
@
\paragraph{Defined Chunks}\par\noindent
\nowebchunks
\paragraph{Index}\par\noindent
\nowebindex
@
\end{document}
```

Listing 2: version exécutable

```
#!/usr/bin/perl
$begin_code_pat = "^\@begin code";
$end_code_pat
= "^\@end code";
$code_line_pat = "^\@text";
$index_prefix = "\@index defn";
sub process_code_chunk {
    while ( ($_ = <>) && !/$end_code_pat/o ) {
        print $_;
        if( /$code_line_pat/o ) {
            $_ =~ s/#.*//o;
```

```
            @def_list = (/sub\s(\w+)/go, /package\s(\w+)/go);
            foreach $item (@def_list) {
                print "$index_prefix $item\n";
            }
        }
    }
    print $_; # make sure we print the ‘‘@end code’’ line
}
while ( <> ) {
    print $_;
    if (/$begin_code_pat/o) {
        process_code_chunk;
    }
}
```

Comme la plupart des outils de programmation lettrée, noweb compte se base sur TEX ou LATEX pour se référer ou mettre en forme la documentation (bien qu'il puisse aussi produire un fichier). Il n'est cependant pas nécessaire d'être un guru en LATEX pour produire de bons résultats car le plus dur du travail est fait automatiquement par noweave.

La documentation mise en forme

Figure 1 le texte mis en forme

Introduction This is `autodefs.perl`[1], a Perl script to be used as an `autodefs` filter in the `noweb` pipeline to identify and index some common Perl definitions. Since this file is also meant to show off some of the features of `noweb` it is purposely verbose and contorted.

Perl does not require the formal declaration or typing of variables which makes it difficult to differentiate between declarations and usages of variables. We may however find definitions of `sub`'s and `package`'s with little difficulty and that is the purpose of this module. Before we begin we need to know some facts about `noweb`'s pipeline structure.[2] Actual code in the pipeline lie between lines of the form `@begin code` and `@end code`. In Perl these are easily recognized by the following regular expressions.

5a ⟨*Global variables* 5a⟩≡ (6b) 5b▷

```
$begin_code_pat = "^\@begin code";
$end_code_pat   = "^\@end code";
```

Defines:
`$begin_code_pat`, *used in chunk 6b.*
`$end_code_pat`, *used in chunk 6c.*

Within a code block there are many types of lines. Ones that contain actual code are prefixed by `@text`.

5b ⟨*Global variables* 5a⟩+≡ (6b) ◁5a 6a▷

```
$code_line_pat = "^\@text";
```

Defines:
`$code_line_pat`, *used in chunk 6c.*

[1] Copyright 1997, Andrew L. Johnson and Brad C. Johnson, All rights reserved.

[2] Noweb's pipeline structure is described in the *Noweb Hackers Guide* which is included in the `noweb` distribution.

If, on a code line inside a code block, we find something that should be added to the "Defines" block at the end of the code chunk and appear in the index, then we need to add a line to the pipeline of the form "`@index defn <ident>`".

6a ⟨*Global variables* 5a⟩+≡ (6b) ◁5b

```
$index_prefix = "\@index defn";
```

Defines:
`$index_prefix`, *used in chunk 7.*

autodefs.perl Our actual Perl script has the following simple shape:

6b ⟨*autodefs.perl* 6b⟩≡

```
#!/usr/bin/perl
⟨Global variables 5a⟩
⟨process_code_chunk subroutine 6c⟩
while ( <> ) {
    print $_;
    if (/$begin_code_pat/o) {
        process_code_chunk;
    }
}
```

Uses `$begin_code_pat` *5a and* `process_code_chunk` *6c.*

Processing the code chunk To process the code chunk we need to perform a few housekeeping tasks. First, we only want to consider lines that begin with `$code_line_pat` and second, we want to stop when we find a line that matches `$end_code_pat`. The following loop will suffice for this purpose.

6c ⟨`process_code_chunk` *subroutine* 6c⟩≡ (6b)

```
sub process_code_chunk {
    while ( ($_ = <>) && !/$end_code_pat/o ) {
        print $_;
        if( /$code_line_pat/o ) {
            ⟨Find and print any definitions 7⟩
        }
    }
    print $_; # make sure we print the ‘‘@end code’’ line
}
```

Defines:
`process_code_chunk`, *used in chunk 6b.*
Uses `$code_line_pat` *5b and* `$end_code_pat` *5a.*

When checking for definitions we first strip off any comments since `sub` or `package` may also occur in a comment. We then build a list `@def_list` which contain all of the `sub` and `package` definitions on the line and print out an `@index defn` line for each.

7 ⟨*Find and print any definitions* 7⟩≡ (6c)

```
$_ =~ s/#.*//o;
@def_list = (/sub\s(\w+)/go, /package\s(\w+)/go);
foreach $item (@def_list) {
    print "$index_prefix $item\n";
}
```

Uses `$index_prefix` *6a.*

Defined Chunks

⟨*autodefs.perl* 6b⟩
⟨*Find and print any definitions* 7⟩
⟨*Global variables* 5a⟩
⟨`process_code_chunk` *subroutine* 6c⟩

Index

Cet exemple montre l'imbrication des morceaux de code réel avec le texte descriptif.

Chaque morceau de code est identifié de façon unique par le numéro de page et une sous-référence alphabétique.

Par exemple, dans la figure 1, il y a quatre gros morceaux codés sur la première page, étiquetés dans la marge 1a, 1b, 1c et 1d.

En plus de l'étiquette marginale, la première ligne de chaque gros morceau codé a aussi un nom et une référence de morceau entourée de crochets en marge gauche avec éventuellement des renvois en marge droite

Voyons de plus près le morceau 1b qui se présente à peu près comme ceci :

1b ⟨*Globalvariables*1*a*⟩+≡ 1d ◁ 1*a* 1*b* ▷

Cette ligne nous dit que nous sommes maintenant dans le morceau 1b.

La construction ’h Global variables 1ai+≡’ nous dit que nous travaillons sur le morceau appelé ’Global variables’ dont la définition commence dans le morceau 1a.

Le '+≡' indique que nous ajoutons à la définition de 'Global variables`.

En marge droite, nous rencontrons ’(1d) ¡1a 1c ¿’, qui signifie que le morceau que nous définissons est utilisé dans le gros 1d et que le morceau actuel continue le morceau 1a et sera continué dans le morceau 1c.

A noter que toutes ces indications de cross-reference sont fournies automatiquement par noweb.

À la fin de n'importe quel morceau il y a deux notes en bas de page optionnelles - une note de bas de page ’Defines’ et une note en bas de page ’Uses’

L'utilisateur peut spécifier manuellement dans le dossier source nw, une liste d'identificateurs (c'est-à-dire de variables ou sous-routines) qui sont définis dans le morceau courant.

De tels identificateurs peuvent être automatiquement reconnus si un filtre 'autodefs' pour le langage de programmation est utilisé (il y a des filtres d'autodefs disponibles pour beaucoup de langages, incluant C, Icon, TEX, yacc et Pascal). Ces identificateurs sont énumérés dans la note en bas de page ’Defines’ au-dessous du morceau où leur définition apparaît avec une référence aux morceaux qui les utilisent.

N'importe quelle occurrence d'un identificateur défini manuellement est référée dans une note en bas de page ’Uses’ au-dessous du morceau qui utilise cet identificateur.

Par exemple, dans la figure 1, nous voyons que le morceau 1c définit le terme $index préfix qui est utilisé dans le morceau 2b.

Un rapide coup d'oeil au gros morceau 2b vérifie cela. le terme est utilisé et apparaît dans la note en bas de page ’Uses’ pour ce morceau.

Le morceau 1d, appelé ‘autodefs.perl’, représente la description de niveau supérieure de notre programme entier.

Ce morceau est considéré comme un morceau ’root’ dans noweb et n'est pas utilisé dans autre gros morceau.

Notre exemple a un seul morceau ’root’, bien que vous puissiez en définir autant que vous voulez dans votre dossier source nw et notangle peut extraire chacun d'eux dans les dossiers séparés.

La première ligne de code dans le morceau 1d est l'obligatoire! ligne

```
#!/usr/bin/perl
```

qui doit commencer tous les scripts perl qui doivent être lancés.

Les deux lignes suivantes ne sont pas des lignes de code de perl mais plutôt des appels à d'autres définitions de morceau. De telles références indiquent que le code des morceaux sera inséré à cet endroit de dans l'exécutable extrait par notangle.

Ainsi nous avons un large aperçu de notre programme sans les encombrantes initialisations de variables globales et de définitions de sous-routines.

En regardant le morceau 2a, qui est inclus dans notre morceau racine, nous voyons que cela inclut aussi un autre morceau, le gros morceau 2b.

Cela montre que l'inclusion de morceaux peut être imbriquée (à pratiquement n'importe quel niveau) et peut apparaître dans n'importe quel ordre dans la documentation (les définitions doivent précéder les utilisations).

Notre documentation finit avec deux indices optionnels fournis par noweb : l'index des morceaux codés et un index d'identificateurs.

L'écriture du Programme dans noweb

Connaissant ce qui sort du pipeline dans la main, nous pouvons aborder la structure du fichier source nw lui-même.

le programme de l'exemple est listé dans la Liste 1.

Quand vous écrivez votre programme noweb, vous alternez entre expliquer certains morceaux de code et la définition formelle de ce code.

Vous devez indiquer si vous entrez dans la documentation ou le code par l'utilisation de deux étiquettes noweb

Pour commencer à écrire la documentation, on commence avec un symbole @ dans la colonne gauche suivie par un espace ou par un retour chariot. Cela indique que tout le texte qui suit, au moins jusqu'à l'étiquette suivante, est du texte de documentation.

Tout le texte de documentation est passé à LATEX par le processus de filtration. L'auteur doit donc fournir les codes de format, comme les sections, les tables, les notes en bas de page et formules mathématiques souhaités dans la documentation.

En plus des commandes LATEX standards, noweb fournit trois contrôles supplémentaires : Un texte entre crochets doubles est formaté comme du code littéral ; et le es commandes de nowebindex et de nowebchunks se développent dans les deux types d'indices montrés à la fin de notre exemple dans la figure 1.

Pour indiquer le début d'un morceau de code, entourez un nom avec « et »= :

```
<<code_chunk_name>>=
```

Ce qui suit cette construction est le code littéral, ou une référence à un autre bloc.

Vous renvoyez à un autre nom de morceau en plaçant son nom entre crochets doubles sans signe égal.

Comme pour la documentation, un morceau de code se termine quand on rencontre une autre étiquette.

Pour continuer une définition de morceau de code, commencez simplement un nouveau morceau de code en utilisant le même nom dans les parenthèses que le gros morceau continué.

Le formatage spécial et le cross-referencing de morceaux de code sont gérés automatiquement par

noweb sans aucune contribution de l'utilisateur - à l'exception des identificateurs spécifiés manuellement par l'utilisateur.

Pour indiquer manuellement une liste d'identificateurs qui sont définis dans un morceau donné, terminez ce morceau par une ligne de la forme :

```
@ %def ident1 ident2
```

The identifiers given on the line will be placed in a ’Defines’ footnote for that chunk and will automatically be cross-referenced and indexed by noweb as described in the previous section.

The process by which notangle extracts the code into a form suitable for the compiler or interpreter follows just a few simple rules. A root chunk is specified on the command line as the chunk to be extracted and assembled. This chunk is then output line by line until a reference to another chunk is encountered. At this point, the referenced chunk is output line by line—and similarly for any chunks referenced therein. When the referenced chunk has been output the process of outputting the root chunk continues.

When dealing with continued chunks—two or more chunks sharing the same name…notangle concatenates their definintions in order of appearance into a single named chunk. The extracted code for our example program is in Listing 2, and it can be seen that all spacing and indentation is preserved appropriately in the executable version.

It is this extraction and assembly process of notangle that allows the ex- planation of the program and the presentation of each part of the program to proceed in an order independent of how the program must be ordered for the compiler or interpreter.

The Incantations.

Now that we know to create a program in noweb we can examine the methods of generating our typeset and executable versions of the program. The noweb dis- tribution provides a general shell script called, remarkably, noweb which drives the notangle and noweave processes. However, this method of invocation, though simple, is somewhat limited. We will focus here on using each tool separately as this provides a more flexible approach. When you invoke notangle you specify a chunk name (a root chunk) to extract and assemble from the nw source file. If you fail to specify a chunk, notangle will search for a chunk named ’*’ to extract (this is the default root chunk in a noweb program). The notangle tool writes to stdout so you must redirect this to a file of your choice. The general form of the command is: notangle [-Rroot_chunk] [-Lformat] [-filter cmd] source.nw > programfile Thus, to extract the executable version of our example program we used: notangle -Rautodefs.perl autodefs.perl.nw > autodefs.perl The -R option specifies which root chunk to extract. The -L option is used to embed line directives, if they are supported by the compiler/interpreter you will be using. The line directives refer to locations in the nw source file, thus, when debugging your code you need not ever refer to the executable version, rather you can simply edit the code in the nw source file. The default format of the line directives is for use with the C preprocessor, but also work well with Perl with one catch. The line directives are emitted whenever a chunk is entered or returned to, and refer to the next line of code. Therefore, in a script such as ours, a line directive winds up as the first line of the executable version, rendering it non-executable. The fix for this is to delete the first line directive, or move it to below the first line and increment the line number by one. One can write filters for use with either notangle or noweave to manipulate the source once in the pipeline. The pipeline representation of the nw source file in noweb is beyond the scope of this article (see the

”Noweb Hacker’s Guide” included in the documentation of the distribution). We will only mention that a filter could easily be constructed which automates the solution to the above mentioned line directive problem. The typeset version of the program is generated with the noweave tool. There are several useful options for noweave, all detailed in the man-pages. Here we will only consider a few of the most important options. The first options of general interest concern the desired output: you may specify -latex (default), -tex or -html as the formatting language to be used for the final documentation. Each of these options will supply an appropriate wrapper (which can be suppressed with the -n option) for the typeset version. You may write your nw source file intended for L A TEXtypesetting and still have the option of producing an html document by invoking noweave with the -html option and the latex-to-html filter (-filter l2h) included with the distribu- tion. The -x option enables cross-referencing and indexing of chunk names and any identifiers which are automatically recognized by an ’autodefs’ filter. Using the -index option implies -x and also provides cross-referencing and indexing for manually defined identifiers—those mentioned in @ %def statements in the nw source file. Normally, noweave will insert additional information such as the filename for use in page headers with its wrapper. The -delay option causes noweave to sus- pend the insertion of this information until after the first documentation chunk. This is most useful when you wish to provide your own (La)TeX wrapper to specify additional packages or defining your own special formatting commands. This implies a -n (omission of wrapper) option and requires that you make sure to include a ’ enddocument’ control sequence in a documentation chunk at the end of the file to complete the wrapper. Our example nw source file is written in this fashion. Our typeset version (Figure 1) was produced by first extracting the au- todefs.perl root chunk with notangle and making it executable with the chmod system command. We then placed this executable in the noweb library directory and invoked noweave as: noweave -autodefs perl -delay -index autodefs.perl.nw > autodefs.tex

This was followed by running L A TEXon the resulting file—twice to resolve page references—to create the dvi file, and then using dvips to create the postscript version for inclusion with this article. Additional options allow you to have the index created from an external file, expand tabs, and to specify alternative formatting options provided by the included noweb.sty file. The latter includes options to omit chunk numbering in the left margins, change text size in code chunks, and switch from using the symbolic cross-referencing of code chunks occurring at the right margin to simple footnote style cross-referencing similar in style to the ’Defines’ and ’Uses’ footnotes.

Pré-requis

Installation

Configuration

Utilisation

Désinstallation

Conclusion

Admittedly, a literate program in general takes a little more time and effort to initially produce. However, as much of this initial effort is devoted to explaining each part of the program, the author is likely to have produced a better quality program in the end because he or she has put more thought into the program's design at each stage of the game. Additionally, by investing in the extra effort of creating a well documented program, the time saved later in maintaining and upgrading the program is considerably lessened. In terms of documentation and explanation, the ability to describe components as they come into play in the design of the program—rather than in the order they must occur for the compiler or interpreter—is a vast improve- ment over traditional commented code. In addition to the benefits of improved code and easier maintanence, literate programs can also serve well as excellent teaching tools.

Availability and Notes

noweb was written by Norman Ramsey, and pointers to obtaining the source and binary distributions of noweb (among other related resources) can be found at his noweb homepage (http://www.cs.virginia.edu/ nr/noweb). The current source distribution contains both awk and Icon versions of the library files necessary. The binary version is built from the Icon source which is recommended as the awk version lacks some of the default behavior of the Icon version. Norman Ramsey has informed us that he is no longer able to maintain and upgrade the awk version. Norman Ramsey has also told us of plans for version 2.8 to include a troff back end (in addition to the TeX, LaTeX and html back ends), conditional tangling, and some pretty printing macros.

Voir aussi

- **(en)** https://www.cs.tufts.edu/~nr/noweb/johnson-lj.pdf
- **(en)** Noweb Hacker's Guide : https://www.cs.tufts.edu/~nr/noweb/guide.html
- **(en)** Noweb example programs : https://www.cs.tufts.edu/~nr/noweb/examples/index.html
- **(en)** Noweb FAQ : https://www.cs.tufts.edu/~nr/noweb/FAQ.html
- page de man de noweb

Basé sur « https://www.cs.tufts.edu/~nr/noweb/onepage.ps » par Norman Ramsey.