Maintaining Cross References in Manuscripts

ALFRED V. AHO
RAVI SETHI

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Authors face the tedious bookkeeping problem of maintaining the consistency of references to figures, citations, and other numbered entities in successive drafts of a manuscript. If a figure is added to or deleted from the manuscript, the numbers of all subsequent figures must be adjusted, along with the references to these figures. In this note, we show how the UNIX™ commands `grep`, `awk`, and `sed` can be used to create a simple and flexible reference assembler that automatically maintains the consistency of cross references in manuscripts.

1. A REFERENCE ASSEMBLER

First, prepare a source text for the manuscript in which each reference is symbolic, such as ‘‘Fig. _Output_’’ or ‘‘Page _Section2_’’, rather than an explicit numeric designation, such as ‘‘Fig. 3’’ or ‘‘Page 3’’. Following Scribe [8], the name of a numbered entity will be called a *tag*, and a symbolic page number a *page label*. Tags represent numbers that are independent of the layout of a manuscript: The number associated with a tag depends on how many other tag definitions precede it in the source text, and can be deduced by looking at the source text alone. Page numbers, on the other hand, depend on the final layout of the text. Section 4 describes a mechanism for handling page labels using the `troff` text-formatter.

In the source text, define each tag by a line of the form

`.@tag` *countervariable tagname*

In a tag definition, `@tag` is a distinctive keyword, *countervariable* is an identifier for an integer variable, and *tagname* is a string used as a tag. The `.` in the first position of the definition indicates that the line is not part of the actual text of the manuscript. Tag definitions and references can appear anywhere in the source text, but the definitions will be numbered in the order in which they appear.

The initial value of a counter variable is zero. Whenever a new tag definition is encountered, the underlying counter variable is incremented and the incremented value of this counter variable is associated with the new tag name. We can create as many different classes of symbolic references as we wish by using a distinct counter variable for each class. In this way, citations, figures, examples, sections, footnotes, and the like can have their own sequence numbers.

Example 1. Figure 1 shows the contents of a file `source` containing the text of a short manuscript. Lines 3, 6, 8, and 11 contain the tags `_Alice_` and `_Huckleberry_`. Lines 7 and 10 contain the corresponding tag definitions

```
.@tag CITE _Alice_
.@tag CITE _Huckleberry_
```

using the counter variable `CITE` to keep track of citation numbers. □

To create the assembled output in which the symbolic references are replaced by numeric ones, execute the UNIX commands in Fig. 2. These commands constitute a complete reference assembler for tags.

The `grep-awk` pipe in the first three lines of Fig. 2 creates in the file `sedscript` a sequence of `sed` instructions that will associate the numbers `1` and `2` with the tags `_Alice_` and `_Huckleberry_`, respectively. Invoked with this script, the `sed` command in the last line of Fig. 2 replaces all occurrences

Source Text

```
.PP
‘‘... ‘and what is the use of a book,’ thought Alice,
‘without pictures or conversations?’ ’’ [_Alice_]
.PP
‘‘... if I’d a knowed what a trouble it was to make a book I
wouldn’t a tackled it and ain’t agoing to no more.’’ [_Huckleberry_]
.@tag CITE _Alice_
.IP [_Alice_]
Carroll, L., Alice’s Adventures in Wonderland, Macmillan, 1865.
.@tag CITE _Huckleberry_
.IP [_Huckleberry_]
Twain, M., Adventures of Huckleberry Finn, Webster & Co., 1885.
```

Fig. 1. Sample source text.

```
grep -h "^\.@tag" source | awk '
          { printf "s/%s/%d/g\n", $3, ++value[$2] }
     END  { printf "/^\\.@tag/d\n" }
' > sedscript
sed -f sedscript source
```

Fig. 2. A reference assembler for tags.

of `_Alice_` and `_Huckleberry_` in the source text by the corresponding numbers and removes the tag definitions from the assembled output.

Example 2. Figure 3 shows the assembled output that would be created by applying the commands in Fig. 2 to the source text in Fig. 1. This assembled output might then be processed with a text-formatter like `troff` or `nroff`. □

Assembled Output

```
.PP
‘‘... ‘and what is the use of a book,’ thought Alice,
‘without pictures or conversations?’ ’’ [1]
.PP
‘‘... if I’d a knowed what a trouble it was to make a book I
wouldn’t a tackled it and ain’t agoing to no more.’’ [2]
.IP [1]
Carroll, L., Alice’s Adventures in Wonderland, Macmillan, 1865.
.IP [2]
Twain, M., Adventures of Huckleberry Finn, Webster & Co., 1885.
```

Fig. 3. The assembled output for the source text of Fig. 1.

2. HOW THE REFERENCE ASSEMBLER WORKS

For readers unfamiliar with the UNIX commands `grep`, `awk`, and `sed`, this section describes how the reference assembler in Fig. 2 works. The assembler makes two passes over the source text. In the first pass, the source text is passed through a pipe consisting of a `grep` command [10] followed by an `awk` command [1] to create a script of `sed` instructions from the tag definitions in the source file. In the second pass, the source program is processed by the `sed` command [5], using the `sed` script created in the first pass, to replace all symbolic references by the corresponding numbers.

Figure 4 depicts what happens when the reference assembler of Fig. 2 is applied to the source text in Fig. 1. The first command

```
grep -h "^\.@tag" source
```

extracts all lines from the source text containing tag definitions, since the symbol `^` in the pattern `^\.@tag` matches the beginning of a line, `\.` matches a period, and `@tag` matches itself. We enclose the pattern in quotes to prevent the UNIX shell from interpreting any characters within the pattern. When `grep` is applied to more than one source file, it normally places the name of the source file currently being processed at the beginning of each output line; the `-h` flag inhibits printing the file name.

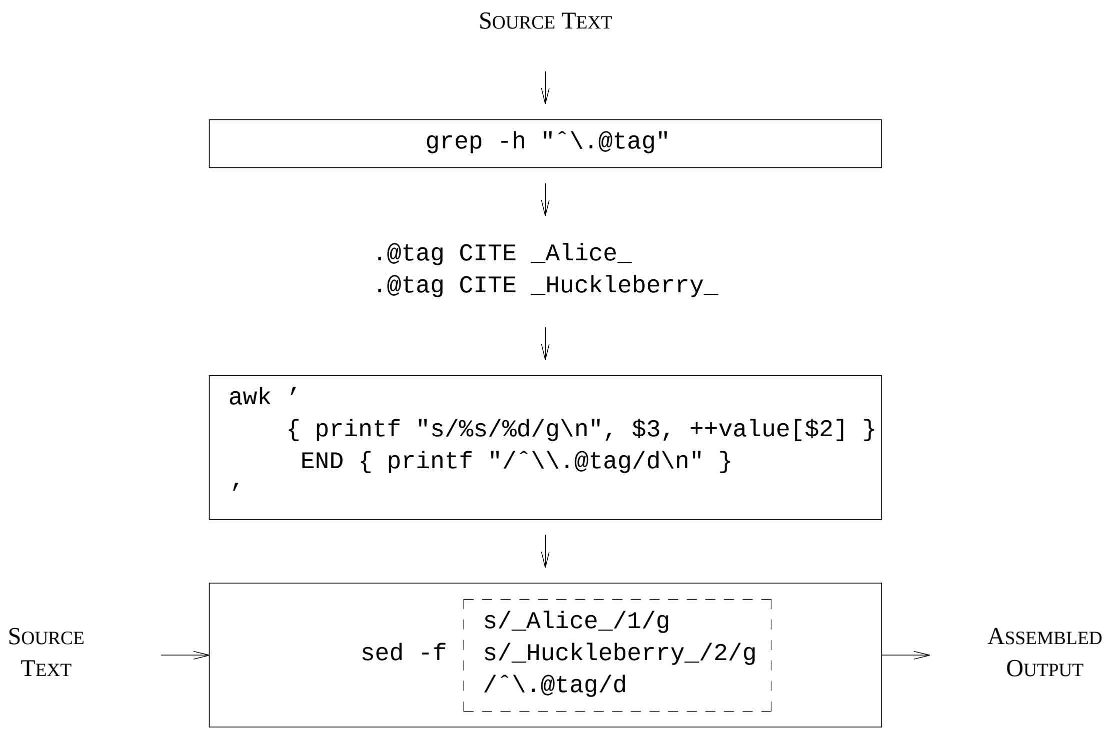

Fig. 4. Reference assembler applied to source text of Fig. 1.

The tag definitions extracted by the `grep` command become the input to the `awk` command, whose output is directed into the file `sedscript`. In the second pass over the source text, `sed` uses the contents of `sedscript` as instructions for replacing tags by numbers.

In the `awk` program, the first statement

```
{ printf "s/%s/%d/g\n", $3, ++value[$2] }
```

is executed on each extracted tag definition. This statement transforms a tag definition

`.@tag` *countervariable tagname*

into a string of the form

`s/`*tagname*`/`*value*`/g`

In the second pass, `sed` uses this string as an instruction to replace each *tagname* in the source text by the corresponding *value*. Since `awk` views each input line as a sequence of fields separated by blanks and/or tabs, the string forming *tagname* is the third field of a tag definition, which is denoted by `$3` in the `awk` program. The second field, denoted by `$2`, is the counter variable. Arrays in `awk` are indexed by strings, so the term `++value[$2]` in the above `awk` statement increments the current value of the counter variable and returns the incremented value. In `awk`, the initial value of every variable is zero.

The `printf` statement associated with the pattern `END` in the `awk` program is executed after all the extracted tag definitions have been processed. This statement prints the string

```
/^\.@tag/d
```

which is used in the second pass by `sed` as an instruction to delete all tag definitions from the source text.

The assembled form of the manuscript is created by processing the source text with the command

```
sed -f sedscript source
```

In Fig. 4, the contents of `sedscript` are enclosed within a dashed box.

Remark. Since the `sed` editing instruction

s/*tagname*/*value*/g

replaces all occurrences of *tagname*, it is helpful to establish notational conventions that prevent *tagname* from inadvertently appearing in other contexts in the source text. The string *tagname* is treated as a pattern by `sed`, so it is best to avoid `sed` metacharacters like `*` within *tagname* [5]. Surprises are also possible if one tag is a prefix of another.

We have used a leading and trailing underscore to distinguish a symbolic name from other text, but any similar convention would work just as well. Note that if we had used `Alice` rather than `_Alice_` as a tag in the source text in Fig. 1, then the lines

```
.IP [Alice]
Carroll, L., Alice's Adventures in Wonderland, Macmillan, 1865.
```

would have been transformed into

```
.IP [1]
Carroll, L., 1's Adventures in Wonderland, Macmillan, 1865.
```
□

Once the editing script has been created from the tag definitions in the complete source text, individual sections of the manuscript can be assembled and formatted independently, if desired.

Example 3. Suppose that the source text in Fig. 1 is in two files `quotes` and `refs`. Then the editing script can be constructed by

```
grep -h "^\.@tag" quotes refs | awk '
          { printf "s/%s/%d/g\n", $3, ++value[$2] }
     END  { printf "/^\\.@tag/d\n" }
' > sedscript
```

We now have the option of editing just the quotes, using

```
sed -f sedscript quotes
```

or the complete source text, using

```
sed -f sedscript quotes refs
```
□

3. MECHANICALLY GENERATED TAGS

Paraphrasing the Scribe Manual [8] slightly, ‘‘the hardest part of using [tags and labels] is in thinking up the code names.’’ The problem of selecting suitable names for tags is compounded if the manuscript has more than one author. Chapter 3 of [2] has over two dozen figures containing finite automata; they could not all be named `_Automata_`.

This problem can be solved by mechanically modifying the tags in the source text to reflect the numbers they represent in the assembled form. A typical mechanically generated tag definition in the source text of [2] looked like

```
.@tag FIG _FIG31_
```

where the tag `_FIG31_` was created from the counter variable `FIG` and its current value `31.` These mechanically generated ‘‘mnemonic’’ tags make the source text (what you see) resemble the assembled output (what you get). If a new tag definition

```
.@tag FIG _New_
```

is inserted into the source text, or if an existing tag definition is removed, fresh tags reflecting the new numbering can mechanically be generated to make the source text resemble the assembled output once again.

The approach in Fig. 2 can be used to create such mnemonic tags. The tags in the original source text are transformed by two successive `sed` scripts, each generated by a `grep-awk` pipe like that in Fig. 2. If the `awk` command in Fig. 2 is replaced by

```
awk '{ printf "s/%s/_@%s%d_/g\n", $3, $2, ++value[$2] }'
```

then the editing script

```
s/_Alice_/_@CITE1_/g
s/_Huckleberry_/_@CITE2_/g
```

is created. Using this script, `sed` creates an intermediate form of the source text with temporary tags `_@CITE1_` and `_@CITE2_`. Then, passing this intermediate form through another invocation of `sed` with the editing script

```
s/_@CITE1_/_CITE1_/g
s/_@CITE2_/_CITE2_/g
```

replaces the temporary tags by the mnemonic ones to produce the desired version of the source text. This two-step approach avoids accidental conflicts if any of the newly created tags happen to look like tags in the original source text. (Some other string can be used instead of `@` to make the temporary tags unique.)

Example 4. Suppose that the order of the two books was reversed in the source text (as would happen in Fig. 1 on page 2 if the books were ordered alphabetically by given name for the authors rather than by pseudonym). The `sed` instructions

```
s/_CITE2_/_@CITE1_/g
s/_CITE1_/_@CITE2_/g
```

followed by

```
s/_@CITE1_/_CITE1_/g
s/_@CITE2_/_CITE2_/g
```

will interchange the tags `_CITE2_` and `_CITE1_`. An attempt to modify the tags in one step using

```
s/_CITE2_/_CITE1_/g
s/_CITE1_/_CITE2_/g
```

will incorrectly replace both `_CITE2_` and `_CITE1_` by `_CITE2_`. □

Tags with common prefixes allow the pattern matching by `sed` to be factored. With the script

```
s/_CITE1_/1/g
s/_CITE2_/2/g
s/_CITE3_/3/g
```

`sed` checks each line for an occurrence of each of the patterns `_CITE1_`, `_CITE2_`, and `_CITE3_` in turn. If the script is rewritten as

```
/_CITE/{
     s/_CITE1_/1/g
     s/_CITE2_/2/g
     s/_CITE3_/3/g
```

```
}
```

then the three patterns are checked only against lines containing the substring `_CITE`. The time to assemble the source files for Chapter 4 of [2] was cut from more than an hour to less than two minutes in this way.

Interrupts, due to second thoughts or due to a power failure, during mechanical modification of the source text can be handled as follows. First, create new files with the modified source text. Second, copy the original source text. Third, replace the original source text files by the modified versions. The original source text is then preserved no matter when an interrupt occurs.

4. REFERENCES TO PAGE NUMBERS

The assembler approach of Section 2 can also be used to maintain references to page numbers, although now we need to make some assumptions about the text-formatting program that does page layouts. In this section we assume page layout will be done using the `troff` text-formatter. In the source text, page labels are defined by lines of the form

`.@label` *pagelabel*

where *pagelabel* is a string used as a symbolic page reference. There is no need for an explicit variable to keep track of page counts.

A page-number assembler is illustrated in Fig. 5. The `grep` command extracts the label definitions from the source text. Suppose, as in the figure, that we are given a table mapping symbolic page labels to educated guesses of the corresponding page numbers. The editing script created by the command $\texttt{awk}_1$ maps page labels to the guessed numbers; the script in the figure would map `_PageA_` to `2` and `_PageB_` to `4`. No guess is available for `_PageC_` so this label is left as is in the source text.

When `troff` formats the source text, we learn the actual page numbers for the labels. A byproduct of formatting is an index table emitted by `troff` into an ‘‘error’’ file separate from the formatted text. Each index-table entry consists of a page label, the actual page number associated with it, and its previous guessed page number. From the index table in Fig. 5, the previous page number was correct for `_PageA_`, incorrect for `_PageB_`, and no information was available for `_PageC_`. The command $\texttt{awk}_2$ checks that the second and third fields of each index-table entry agree, and if they don’t, warns the user of the inconsistent information. The index table created on one formatting pass becomes the table of guesses for the next pass. If the new page numbers do not agree with old ones in the index table, we reformat the source text using the new page numbers. As soon as the new page numbers agree with the old, we know we have a correct page numbering. A possible user interface is marked by dotted lines.

The editing script created by the command $\texttt{awk}_1$ replaces each line

`.@label` *pagelabel*

in the source text by a line

`.@label` *guess pagelabel*

Occurrences of *pagelabel* everywhere else in the source text are replaced by *guess*. The editing script for the example in Fig. 5 is

```
/^\.@label/{            do the following on lines starting with .@label
     s/ .*/&&/            replace .@label pagelabel by .@label pagelabel pagelabel
     s/_PageA_/2/         replace first copy of _PageA_ by 2
     s/_PageB_/4/         replace first copy of _PageB_ by 4
}
/^\.@label/!{           do the following on lines that do not start with .@label
     s/_PageA_/2/g        replace all copies of _PageA_ by 2
     s/_PageB_/4/g        replace all copies of _PageB_ by 4
}
```

The index table can be created by using `troff` instructions of the form

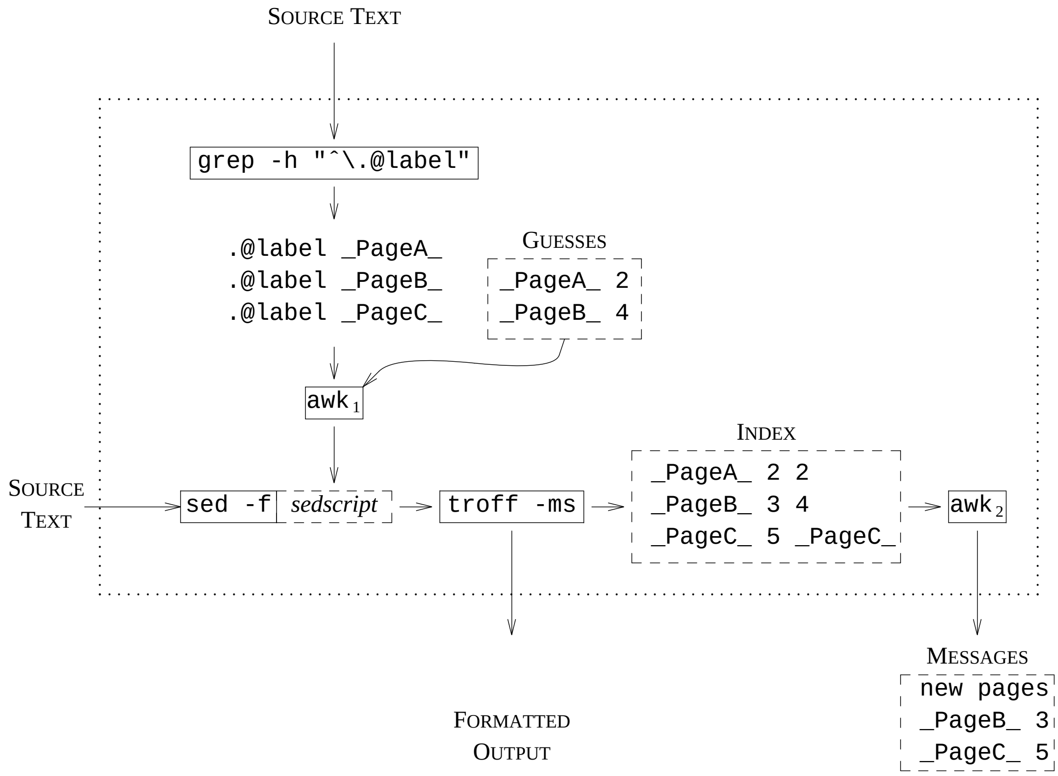

Fig. 5. Page-number assembler.

`.tm` *string*

This instruction emits *string* to the standard error file. The `troff` number register `%` holds the current page number. Its value replaces the string `\n%` in the output, so the `troff` instruction

```
.tm _PageB_ \n% 4
```
(1)

emits `_PageB_` and its new and old page numbers.

This simple approach works only if the `.tm` command is printed immediately on the current page. If the command appears in a ‘‘keep’’ or a footnote that spills onto the next page, then `\n%` will incorrectly be replaced by the page number when the `.tm` command is first seen, rather than the page number when the block of text containing the command is printed.

There is a way to determine whether a block of text is about to be printed in `troff`: test if register `.z` contains the empty string. The command

```
.if '\n(.z'' .tm _PageB_ \n% 4
```
(2)

executes the `.tm` command if register `.z` contains the empty string; otherwise no action occurs. For text that might be squirreled away and reread some number of times by `troff`, the test can be done recursively, each time the text is reread. The following macro `.@l` can be used; (`troff` interprets

```
.@label 4 _PageB_
```

as the application of macro `.@l` to three arguments: `abel`, `4`, and `_PageB_`).

```
.de @l
.ie '\\n(.z'' .tm \\$3 \\n% \\$2
.el \\!.@label \\$2 \\$3
..
```

Here, `.ie` corresponds to `.if` in (2) and `.el` is the ‘‘else’’ part containing the recursive call to `.@l`. The `\\!` before `.@l` delays the recursive call until the text is reread. The `ms` macro package [3] may reread text upto three times.

Appendix A contains the code for an assembler that maintains symbolic references to both tags and page labels.

5. CONCLUDING REMARKS

The main point of this note is to show that the problem of maintaining cross references can be flexibly solved with a few lines of `grep`, `awk`, and `sed` code. In addition, when dealing with tags, the problem can be solved independently of text formatting. The reference assembler described in Section 1 does not even need to know that the source text will be formatted. An assembler of this nature was used to maintain cross references to the more than one thousand numbered algorithms, equations, examples, exercises, figures, and footnotes in the successive drafts of the source text for [2].

Facilities for resolving cross references to tags and page labels have appeared before in text-formatters. Reid [7] mentions that Scribe’s cross-referencing facility is styled after the counter and referencing mechanism used in Pub [9]. McIlroy [4] relates that around 1972, ‘‘One clever Roffer did the job by Roffing a whole document with printing turned off, gathering cross references, then doing it again with printing turned on . . . This is all very well, except that depending on the size of the numbers inserted, subtle changes in pagination of the document could happen the second time through.’’

Miller [6] has implemented a cross-reference assembler in C.

References

[1] AHO, A. V., B. W. KERNIGHAN, AND P. J. WEINBERGER [1979]. ‘‘Awk – a pattern scanning and processing language,’’ *Software—Practice and Experience* **9:4**, 267-280.

[2] AHO, A. V., R. SETHI, AND J. D. ULLMAN [1986]. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass.

[3] LESK, M. E. [1976]. ‘‘Typing documents on the UNIX system: using the `-ms` macros with `nroff` and `troff`,’’ Computing Science Technical Report 49, AT&T Bell Laboratories, Murray Hill, N.J. See *UNIX Programmer’s Manual* Volume 2.

[4] MCILROY, M. D. [1973]. ‘‘Roff text formatter,’’ MHCC-005, Revision 1, AT&T Bell Laboratories, Murray Hill, New Jersey.

[5] MCMAHON, L. E. [1979]. ‘‘Sed – a noninteractive text editor,’’ Computing Science Technical Report 77, AT&T Bell Laboratories, Murray Hill, N.J. See *UNIX Programmer’s Manual* Volume 2.

[6] MILLER, C. D. F. [1985]. ‘‘Lbl – symbolic labels in text documents,’’ manuscript, Heriot-Watt Univ., Edinburgh, Scotland.

[7] REID, B. K. [1980]. ‘‘Scribe: a document specification language and its compiler,’’ Report CMU-CS-81-100, Carnegie-Mellon Univ., Pittsburgh, Pa.

[8] REID, B. K. AND J. H. WALKER [1980]. *Scribe User’s Manual*, Third Edition, Unilogic Ltd., 605 Devonshire St., Pittsburgh, Pa. 15213.

[9] TESLER, L. [1972]. ‘‘Pub: the document compiler,’’ Report ON-70, Artificial Intelligence Project, Stanford Univ.

[10] THOMPSON, K. [1968]. ‘‘Regular expression search algorithm,’’ *Comm. ACM* **11:6**, 419-422.

APPENDIX A

NAME

`xref` — create sed script to resolve cross references

SYNOPSIS

`xref` *files*

DESCRIPTION

`xref` reads *files* containing lines of the form

`.@tag` *counter tag*
`.@label` *pagelabel*

and an optional file `xref.index` mapping page labels to expected page numbers. It creates a `sed` script to replace tags and page labels by numbers. Label definitions are replaced by `troff` commands to emit lines of the form

`.@index` *pagelabel pagenumber oldpagenumber*

destined for file `xref.index`

Typical usage is

```
xref files >.script
sed -f .script files |pic|tbl|eqn| troff -ms >outfile 2>xref.index
```

The command

```
awk '
  $3!=$4 { change = change sprintf("%s %s\n",$2,$3);
         }
  END    { if (change != "")
               system( sprintf("cat 1>&2 <<\!\nnew pages\n%s!\n", change) );
         }
' xref.index
```

warns of changed entries in the index.

FILES

`xref.index`	index mapping page labels to page numbers
`xref.tmp`	temporary file

SEE ALSO

`sed`(1)

DIAGNOSTICS

Detects reuse of a name as a tag or label.

CODE

Tags alone are much easier to handle than page labels. Delete the commands between lines beginning with ## to restrict the code to tags.

```
#    Extract tag and label definitions.
grep -h "^\.@" $* >xref.tmp

#    Create commands for replacing tags, and check for redeclarations.
awk '
 $1==".@label" { check($2, $1);
```

```
                }
 $1==".@tag"   { check($3, $1);
                 printf "s/%s/%d/g\n", $3, ++value[$2];
               }
 END           { printf "/^\\.@tag/d\n"
               }
 func check(name,cmd) {
    if (seen[name])
        system( "echo "name" redeclared in "cmd" 1>&2" );
    else seen[name] = 1;
 }
' xref.tmp

##   Create commands for replacing page labels.
#    Delete up to just before "rm xref.tmp" if tags only are needed.
cat <<\!
1i\
.de @l\
.ie '\\\\n(.z'' .tm .@index \\\\$3 \\\\n% \\\\$2\
.el \\\\!.@label \\\\$2 \\\\$3\
..
/^\.@label/{
     s/ .*/&&/
!
if test -f xref.index
then {
  awk '
    $1==".@label" { seen[$2] = 1;
                  }
    $1==".@index" { if (!seen[$2]) continue;
                    s1 = sprintf("%s\ts/%s/%d/\n", s1, $2, $3);
                    sg = sprintf("%s\ts/%s/%d/g\n", sg, $2, $3);
                  }
    END           { printf "%s}\n", s1;
                    printf "/^\\.@label/!{\n%s\n", sg;
                  }
  ' xref.tmp xref.index
}
fi
echo "}"
##   Delete up to here if tags only are needed.

rm xref.tmp
```

BUGS

Special characters in tags and page labels are interpreted by `sed`; tags and labels can safely be made up of letters, digits, and underscores.