ELECTRONIC PUBLISHING, VOL . 1(1), 3–17 (APRIL 1988)

Tools for printing indexes

JON L. BENTLEY AND BRIAN W. KERNIGHAN
AT & T Bell Laboratories
600 Mountain Avenue
Murray Hill
NJ 07974
USA

SUMMARY

This paper describes a set of programs for processing and printing the index for a book or a manual. The input consists of lines containing index terms and page numbers. The programs collect multiple occurrences of the same terms, compress runs of page numbers, create permutations (e.g., 'index, book' from 'book index'), and sort them into proper alphabetic order. The programs can cope with embedded formatting commands (size and font changes, etc.), with roman numeral page numbers, and with *see* terms. The programs do not help with the original creation of index terms.

The implementation runs on the UNIX® operating system. It uses a long pipeline of short *awk* programs rather than a single program in a conventional language. This structure makes the programs easy to adapt or augment to meet special requirements that arise in different indexing styles. The programs were intended to be used with *troff*, but can be used with a formatter like TEX with minor changes.

An appendix contains a complete listing of the programs, which total about 200 lines.

KEY WORDS indexing awk Unix troff document preparation

MAKING AN INDEX

There are two major tasks to making an index for a book or manual. The first is deciding on the proper indexing terms, so that users of the index can readily find what they are looking for. This is hard intellectual work if done well, and no mechanical aid is likely to do more than help with a rough first draft. The authors of this paper have between them indexed nearly a dozen books and as many programmer manuals, and have never found a substitute for a lot of thought. (Reference[1] contains an excellent discussion of the entire process of producing an index by traditional means.)

The second task is, given a set of terms and page numbers, to produce and print a properly sorted and formatted index. This includes collecting multiple instances of an index item into a single list of page numbers:

book index 1, 17, 18, 19, 25, 26

permuting index terms:

index, book 1, 17, 18, 19, 25, 26

compressing runs of adjacent page numbers:

book index 1, 17–19, 25–26

0894–3982/88/010003–15$07.50

Received 15 October 1987
Revised 25 December 1987

sorting correctly in the face of strange characters and formatting commands:

> `ps -a` 34, 91
> `ps` command 34
> `.ps` command, `troff` 301
> `PS1` shell variable 36, 82

and a host of similar details.

This second task—mechanical but remarkably time-consuming if not mechanized—is addressed by the family of programs described here.

The precise task to be performed depends on the style of the index. Some issues are cosmetic: for example, which of the following styles is desired?

> index term, ii, iii, 26.
> index term ii, iii, 26
> index term ii–iii, 26

Other issues are deeper. This example incorporates a glossary, allows hierarchical entries, and includes *see* and *see also* cross-references:

> Insertion: adding a new element, 168-223.
> into arrays, 169.
> into binary trees, *see* Trees.
> into linked lists, 200–215, *see also* Sequences.

How should a program deal with such a multitude of choices? One way is to build a number of options into a large program, controlled perhaps by various flags. This approach solves many problems, but it requires a great deal of complex code, since the various options have subtle interactions. It is also difficult to modify such code if one wishes to cater to problems not addressed in the original.

We have taken an alternate approach to the problem of proliferating options. Our package provides basic services—permuting terms, compressing runs, sorting correctly—but neglects formatting options, glossary definitions, hierarchies, and cross references. The simple package is sufficient for producing simple indexes. Users with more complex needs must modify the programs; the tools are organized as a long pipeline of short *awk* programs to make this easy.

The programs run on the UNIX operating system. They are intended for use with the *troff* formatter, but can be readily adapted to another formatter, such as TEX. The reader is assumed to have some familiarity with UNIX.

DESIGN CONSIDERATIONS

The first step faced by someone making an index is to prepare a list of index terms and page numbers. This can be done completely by hand if the document is guaranteed to be in its final form, by transcribing the terms and page numbers into a file. Another possibility is to make a list of index terms by scanning the document mechanically. This is difficult; no successful automatic indexing programs are known to the authors.

The most satisfactory way to make a list seems to be to include, in the machine-readable form of a document, commands that cause the index terms and their computed page numbers to be emitted when the document is formatted. For example, with the *troff*

formatter, one defines a macro named `.ix`, and inserts a macro call for each phrase to be indexed:

```
This paper describes a set of programs for processing
and printing the index for a book
.ix book index
or a manual.  The input is ...
```

With the LATEX package of TEX macros[2], one would say instead:

```
This paper describes a set of programs for processing
and printing the index for a book\index{book index}
or a manual.  The input is ...
```

As the document is formatted, the index output is captured, then processed by the indexing programs to create sorted data ready to be formatted. The index is printed by a subsequent execution of the formatter.

Although adding terms to a document is not difficult, it is important to make it as easy as possible, since there might easily be half a dozen terms on every page. Thus operations like creating permutations of terms should be automatic, and any control information should be compact. In our system, a small language is used in index terms to control phrase permutation, font changes, and sorting order. The most important criteria for this language have been simplicity and conciseness.

The simplest form of an indexing entry is:

```
.ix sorting a book index
```

which, if it occurs on page 97, will be output as:

```
sorting a book index tab 97
```

Permutations are generated automatically; this phrase will subsequently be converted into four entries by rotating it around each blank:

```
sorting a book index            97
a book index, sorting           97
book index, sorting a           97
index, sorting a book           97
```

There must be a way to control the automatic rotation so that rotations are generated only as desired. The simplest and most compact notation is to use one character that will sort and print as a blank but still prevent rotation. We chose the character ~ for this function:

```
.ix sorting~a book~index
```

will eventually produce:

```
book index, sorting a   97
sorting a book index    97
```

If there are multiple occurrences of an indexing phrase on adjacent pages, they may be collected and merged to appear as, for example, 97, 98, 108–110. The specific choice (combining three or more consecutive pages but not two, for example) is an instance of an option that different users will want to control. Rather than providing flags or

parameters for control, such variations are obtained by making (generally trivial) modifications to the programs. Our programs combine a sequence of consecutive page numbers into a hyphenated pair, but variations such as combining only sequences of length three or more are obtained by changing one line in one function (`printoldrange` in the program `range.collapse`).

Not all page numbers are arabic. Books use Roman page numbers for front matter; numbers in lower case roman numerals (i, ii, iii, ...) are generally sorted before arabic page numbers. Again, rather than writing code to take care of a family of possibilities, we have isolated the processing of roman numerals in two short programs (`deroman` and `reroman`) which can be adapted to handle alphabetic page numbers like A-1 as well.

Our indices have dealt largely with programming topics, where typographic convention is to use fonts like `monospace` and *italic* for keywords, variables, program names, and the like. When such terms are to appear in indices, special care must be taken to have them sorted properly. The sheer number of such terms also mandates a concise input. (The index to Reference[3] has 1015 index terms, of which 460 involve a font change.) Bare formatting requests are too bulky in *troff* and not much better in TEX. Accordingly the language provides two abbreviations for font changes: a part of a term enclosed in `[]` is printed in `monospace` and a part enclosed in `{}` is printed in *italics*. These choices are of course arbitrary, and easy to change or drop entirely.

As soon as such conventions are used, then there has to be a way to include a literal occurrence of a metacharacter like `~` or `[` in an index term. We chose to use the single character `%` as the quoting character, since it occurs infrequently, but again the specific character is easy to change.

These three examples illustrate the use of font change commands and quoting:

```
.ix [pr]~[-]{n} command
.ix [%[^]...[%]] regular~expression
.ix [printf] [%%d] specification
```

These will appear in the final index as

%d specification, printf
[^...] regular expression
command, pr -*n*
pr -*n* command
printf %d specification
regular expression, [^...]
specification, printf %d

Sorting is normally performed with the index term as the sort key. The control commands listed above are removed from the sort key so they do not affect the order of sorting, as are *troff* font changes like `\f(CW` and `\fI` and size changes like `\s8` and `\s-3`.

Some terms are so complicated or special that they cannot be sorted directly. The most typical examples involve embedded formatting commands, such as the *troff* string `T\v'.17m'\h'-.12m'E\h'-.12m'\v'-.17m'X` which prints as TEX. To deal with this, the control language provides for specification of an explicit sort key. The sort order may be controlled by adding a sort key to an index term, with `%key`:

```
.ix any string %key explicit sort key
```

as in:

```
.ix T\v'.17m'\h'-.12m'E\h'-.12m'\v'-.17m'X %key TEX
```

This mechanism handles many situations, but not everything. For instance, where do operators like `+` or `**` sort? What about numeric values like `68000`? Our program arranges that non-alphanumerics sort first, then numbers, then text, but it is a trivial change to have special characters ignored during sorting.

An entry like '*large subject 19-35*', to indicate a range of pages, is specified with two special `.ix` entries, `%begin` and `%end`:

```
.ix %begin large subject    .. on the first page
.ix %end large subject      .. on the last page
```

To summarize the language that controls formatting and sorting of indexing terms:

`~`	prints as blank, but causes no rotations
`[...]`	will print ... in `monospace font`
`{...}`	will print ... in *italics*
`%~`	literal ~
`%%`	literal %
`%e`	printable troff escape character \
`%[, %]`	literal [,]
`%{, %}`	literal {, }
`%key`	explicit sort key follows
`%begin`	start a range of page numbers
`%end`	end a range

If any character not mentioned above follows the escape character %, it is printed as that literal character.

There are still many features that might be desirable—*see* and *see also* references, hierarchical terms, and the like. A later section discusses the addition of a *see* cross-reference facility, and suggests how other features might be added.

IMPLEMENTATION

The indexer consists of a set of small *awk* programs, intentionally kept separate for easy modification. (The appendix contains a complete listing of all of the programs.) For example, roman numeral page numbers are processed by `deroman` and `reroman`. These programs can only count up to *xxx*, however, so a change is needed to count beyond 30. On the other hand, if there are no roman-numeral pages, `deroman` and `reroman` are unnecessary.

The basic strategy is to sort once to bring together all occurrences of identical index terms so as to combine their page numbers. Preprocessing is done first to deal with roman numeral pages and with any `%begin`/`%end` entries. After sorting, roman numerals are restored and runs of page numbers are combined.

The next step is to generate rotated entries; each generated term of a group carries the same list of page numbers.

The next step is to sort the terms into final order. Correct sorting in the face of bizarre

font controls and the like is achieved by prefixing a sort key to each line such that sorting on that key creates the proper order; the `%key` command allows for overriding of the default sort key.

The final step is to expand font change commands and ~'s. This step also inserts a *troff* macro call before each entry and before the first entry for each letter of the alphabet. The definitions of these macros (in the file `index.head`) control the format of index terms in the printed version—point size and leading, number of columns, and the like.

The whole process is controlled by a shell file `make.index`, which produces the body of the index file:

```
make.index ix.raw ... >index.body
```

The resulting body is typeset into an index in a subsequent step. Using *troff*, for instance:

```
troff -ms index.head index.body >index.out
```

The appendix contains a sample `index.head` file.

The specific programs in `make.index` are, in order,

`doclean`	strip excess spaces, remove any non-index lines
`deroman`	map roman numerals to arabic
`range.prep`	prepare to sort (handle `%begin`/`%end`)
`range.sort`	sort by string then page number
`range.collapse`	resolve `%begin`/`%end` and merge runs of page numbers
`reroman`	put arabic numerals back into roman
`num.collapse`	put many number pairs onto one line
`rotate`	make rotated copies of each line
`gen.key`	generate a sort key, if one wasn't provided
`final.sort`	sort using the key
`format`	do font and size changes, etc.

The *awk* programs rely on features in the *awk* interpreter released in mid-1985[3]. The 'pipeline' actually uses temporary files to connect the stages (since some systems have a small limit on the number of programs in a pipeline and the intermediate files are also useful for debugging), and deletes the files at the end.

BELLS AND WHISTLES

Our programs produce a basic index. Additional features are built by adding to or adapting them. To illustrate the process, we consider a simple addition: '*see*' references of the form:

secondary term *see* primary term

(The examples in the introduction also illustrate a *see also* reference to follow a list of page numbers; we shall leave that as an exercise.) The *see* references must be placed in the file `see.terms` in the format:

secondary term *tab* primary term

The general strategy is to have an *awk* program massage that file into a suitable format, then merge it into the existing pipeline.

Here are the implementation details. The `make.index` command is replaced by:

```
doclean ix.raw | deroman | range.prep | range.sort |
        range.collapse | reroman | num.collapse |
        rotate | gen.key | final.sort >junk.regular
sort see.terms | see.prep >junk.see
sort -mfd junk.see junk.regular | format >index.body
```

The *see* terms are sorted, processed by `see.prep`, then merged into the larger file by `sort`'s `-m` option. The program `see.prep` requires two lines of *awk*:

```
awk ' # see.prep
BEGIN { FS = "\t" }
      { print $1 "\t" $1 "\t\\fIsee\\fP " $2 }
' $*
```

This program uses the secondary term as the sort key and as the term itself; it puts '*see* primary term' in the third field (which is normally occupied by page numbers). This assumes that the terms contain no formatting commands; if they do, they must be piped through `gen.key` as well.

The most obvious missing piece is hierarchical indexes, which can be arbitrarily complex:

book
 composition 23
 indexing 45–54
  automatic 50
  manual 48
 production 67

Our tools do not supply hierarchies because the indexes in our books don't use them (or *vice versa*). Rather, we achieve a similar effect by careful use of rotation of phrases:

search 1–10, 12–14, 140–148
search, binary 12–13, 16, 18
search, hash 90, 121, 142–143, 145–146
search, sequential 12, 18, 46

The next easiest step is a two-level hierarchy, in which the primary and secondary keys are explicitly identified in the input, perhaps as:

```
.ix primary term %, secondary term
```

(More than two levels is hard and not too useful; deducing primary and secondary keys from word strings is a hit-and-miss operation.) Implementation requires changing `format` and the `index.head` file, and writing a new macro for secondary terms.

The programs do not deal directly with complicated material like mathematics in index terms. As it stands, these can be handled best by explicit sort keys, which is probably adequate if there are not too many such items.

Although our tools were designed to work with *troff*, it is straightforward to adapt them to other document production systems, such as TEX. The first part of the job is to produce an analog of the `.ix` macro to emit index terms and page numbers. For example, `\index{`*term*`}` in LATEX is essentially identical to our `.ix` macro. `doclean`

must be modified to sweep up any loose ends, and `format` changed to produce output of the right form; the rest of the pipe is unchanged. The resulting output is incorporated into the document by a mechanism like `index.head`.

We also use the `.ix` macro to generate text and page numbers for tables of contents. A macro for producing section headings, for instance, might be augmented to produce lines of the form:

```
.ix CONTENTS Section Number Section Title
```

A subsequent program separates table-of-contents items from index terms and prepares them in a format suitable for *troff*. (This is why `doclean` filters out lines that contain the string '`CONTENTS`'.)

As a final observation, indexing is often done late in the game, under intense time pressure. In such circumstances, there is no disgrace in using a text editor (as a last resort) to fix up things that just don't work correctly.

COMMENTS ON PROGRAMMING STYLE

This is the third version of a family of indexing programs started more than a decade ago. The first and second versions used a pipeline of C programs, and increasingly complicated *sed* scripts and *sort* options, in a largely unsuccessful attempt to control sorting order; they are sketched in the index of Kernighan and Plauger[4]. As capabilities were added over the years, the programs degraded into write-only code, comprehensible only with substantial effort.

Our motivation for building the current set of *awk* programs was to prepare the index to Bentley's book[5]. This index was substantially different from those processed by the existing programs: it had no font changes (which had contributed greatly to the complexity of the C programs), but it did employ other niceties, such as ranges of pages and breaks between letters. Rather than modifying the existing suite, we spent a few hours building a single-shot *awk* pipeline for the task (37 lines of *awk* in 6 programs).

Several months later we built the current version, which is a functional superset of its two predecessors. We worked with a colleague who was preparing the index to a manual, and to whom we had described the prototype. We wrote the new code, debugged it, added several necessary features, and provided initial documentation, all within a week. The code presented here is a modest improvement on that. (There is also a short version in Reference [3].) The final version of the C program, the initial *awk* program, the textbook *awk* program and the final *awk* program are summarized in Table 1.

Table 1 has been massaged to compare incomparables. The C version, for instance, did not have separate programs for `deroman` and `reroman`; it performed those tasks in its *sed*-script versions of `gen.key` and `format`, so we redistributed the line counts. The C version did not support ranges, which were entered explicitly by the user in the first *awk* system, so neither prototype had the three programs that compute ranges. The prototype *awk* programs performed no font changes, but were careful with roman numerals.

The final *awk* suite is six times longer than the prototype, due to improvements in several important dimensions.

Table 1. Lines of source code

PROGRAM	C PROTOTYPE	AWK PROTOTYPE	AWK TEXTBOOK	AWK PRODUCTION
`doclean`	3 sed			11
`deroman`	7 sed	7		17
`range.prep`				9
`range.sort`		1 sh	4 sh	4 sh
`range.collapse`			10	23
`reroman`	4 sed	10		22
`num.collapse`	49 C	4		12
`rotate`	60 C	6	8	16
`gen.key`	18 sed		9	39
`final.sort`	1 sh	1 sh	1 sh	4 sh
`format`	30 sed	10	7	47
Total Lines	172	37	36	204.SP .3

Functionality: The programs support computed ranges, font changes, and several other additions.

Error-checking: An error message is produced when a range was started but not ended, for instance.

Bomb-proofing: Sanity is maintained for a wide class of invalid inputs, such as huge roman numerals.

Performance: Improvements, ranging from more sophisticated algorithms to *awk* by an order of magnitude.

Readability: Although this may strain the imagination of some readers, the first version was much less readable than the code presented in the appendix. (Fifty of the 200 lines are comments.)

These issues were not important in a single-shot prototype, but do matter in a production program. Improving the C prototype might well also increase its length by a factor of six.

The essence of the final suite is a long pipeline of short *awk* programs. Is that a good approach? A pipeline proved to be an effective decomposition for this task: each program follows the pipe philosophy of performing one task well, and is only slightly muddled by the format of its input and output. We are familiar with two monolithic programs for producing indices. One is for its author's personal use for *troff* books; it is 350 lines of prototype-quality C and makes use of several system utilities. The other is the TEX indexer described by Chen and Harrison[6]. It is a single program, implemented in 7800 lines of C. The program handles a wide variety of indexing options, such as hierarchical terms, *see* references, page ranges, and bold page numbers. If our suite of programs handles 90 percent of the indexing job, then their program handles 99 percent. The next 9 percent cost a factor of almost 40 in code size. If one desires an option in the remaining 1 percent not covered by Chen and Harrison's system, it might be easier to modify something smaller.

Fragmenting the job into a large number of small pieces makes it easy to add or change pieces; this seems especially important for indexing, where there is a wide variety of styles. It also leaves open the possibility of recoding some critical part for speed. (For

a discussion of decomposition strategies applied to making a KWIC index, a simpler problem, see [7].)

The *awk* language is better suited than C to the combination of string handling, pattern matching and arithmetic. There is an order-of-magnitude difference between lines of C code and lines of *awk* code for the prototype versions of `num.collapse` and `rotate`; this ratio appears to be typical for tasks of this nature. The shorter *awk* version is also much closer to being correct. As the authors can attest from personal experience, however, any automatic process can create insidious errors if its output is accepted blindly.

Performance does suffer in the *awk* version. The index to Reference [3] (1015 index terms) takes 11 seconds with the old C version and 23 seconds with the new one on a VAX-8550. A similar ratio holds for other indices. This factor of two seems acceptable, however, for a program that is run only occasionally and is fast enough for most users.

ACKNOWLEDGMENTS

We are grateful to Al Aho, Ted Kowalski, Doug McIlroy, Ravi Sethi, Chris Van Wyk, Pamela Zave, and anonymous referees for comments on this paper. Pamela Zave also gave us much help with shaking down the current program. Ravi Sethi provided the `.ix` macro in the appendix.

REFERENCES

1. *Words into Type*, Prentice-Hall, 1974.
2. Leslie Lamport, *LATEX: A Document Preparation System*, Addison-Wesley, 1986.
3. Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, *The AWK Programming Language*, Addison-Wesley, 1988.
4. Brian W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.
5. Jon L. Bentley, *Programming Pearls*, Addison-Wesley, 1986.
6. Pehong Chen and Michael A. Harrison, 'Issues in Index Preparation and Processing', Unpublished Manuscript, University of California, Berkeley (1987).
7. David L. Parnas, 'On the criteria to be used in decomposing systems into modules', *Communications of the ACM*, **15** (12), 1053–1058 (1972).

APPENDIX: THE PROGRAMS

This appendix lists the programs verbatim in the order in which they are used. The programs are available in machine-readable form by sending the mail message:

```
send indexing.tools from typesetting
```

to either <netlib@anl-mcs.arpa> or <research!netlib>.

ix.macro:

```
.de ix
.ie '\\n(.z'' .tm ix: \\$1 \\$2 \\$3 \\$4 \\$5 \\$6 \\$7 \\$8 \\$9      \\n%
.el \\!.ix \\$1 \\$2 \\$3 \\$4 \\$5 \\$6 \\$7 \\$8 \\$9
..
```

index.head:

```
.\" This version is for the -ms macro package
.pn 999          \" page number for first page
.de XX           \" this macro precedes each index term
.br              \"   break
.ti -.2i         \"   outdent first line of each entry
.ne 2            \"   need two lines for typical entry
..
.de YY           \" header between letters of the alphabet
.sp 1.5          \"   space 1.5 lines
.ne 3            \"   need 3 lines on this page
.ce              \"   center next output line
- \\$1 -         \"   print the letter
.sp .5           \"   space .5 line
..
.SH              \" provide heading
Index
.LP              \" text is coming
.nr PS 8         \" index looks better in small type
.nr VS 9         \"   and small spacing
.MC 1.9i         \" 3 columns with default-size page
.na              \" no-adjust gives ragged right lines
.in .2i          \" outdent first line by 0.2 inches
.hy 0            \" don't hyphenate
```

make.index:

```
# make.index: prepare an index from $*
doclean $* >foo0
 deroman foo0 >foo1
  range.prep foo1 >foo2
   range.sort foo2 >foo3
    range.collapse foo3 >foo4
     reroman foo4 >foo5
      num.collapse foo5 >foo6
       rotate foo6 >foo7
        gen.key foo7 >foo8
         final.sort foo8 >foo9
          format foo9
# rm foo[0-9]           # comment out for debugging
```

doclean:

```
awk ' # doclean
#   Input:  string (blanks and tab) number
#   Output: string (tab) number
BEGIN         { FS = OFS = "\t" }
$0 !~ /^ix: / { print "doclean: non index line: " $0 | "cat 1>&2"; next }
/CONTENTS/    { next } # some people use .ix to make a table of contents
              { sub(/^ix: /, "", $1) # rm leading "ix: "
                sub(/ +$/, "", $1)   # rm trailing blanks
                print
              }
' $*
```

Piping the output of a `print` statement through `cat 1>&2` is an *awk* idiom for sending output to the standard error.

deroman:

```
awk ' # deroman
#   Input:  string (tab) [arab or roman]
#   Output: string (tab) [arab]
#       Roman numeral n is replaced by arab n-1000 (e.g., iii -> -997)
BEGIN           { FS = OFS = "\t"
                  # set a["i"] = 1, a["ii"] = 2, ...
                  s =   "i ii iii iv v vi vii viii ix x"
                  s = s " xi xii xiii xiv xv xvi xvii xviii xix xx"
                  s = s " xxi xxii xxiii xxiv xxv xxvi xxvii xxviii xxix xxx"
                  n = split(s, b, " ")
                  for (i = 1; i <= n; i++) a[b[i]] = i
                }
$2~/^[ivxlc]+$/ { if ($2 in a) $2 = -1000 + a[$2]
                  else print "deroman: bad number: " $0 | "cat 1>&2"
                }
                { print }
' $*
```

This program uses *awk*'s strings and `split` command to initialize the array `a`; this idiom occurs in several later programs.

range.prep:

```
awk ' # range.prep
#   Input:  [istart/iend] string (tab) number
#   Output: string (tab) [b/e] (tab) number
BEGIN           { FS = OFS = "\t" }
                { f2 = "" }
$1 ~ /^%begin/  { f2 = "b"; sub(/^%begin */, "", $1) }
$1 ~ /^%end/    { f2 = "e"; sub(/^%end */, "", $1) }
                { print $1, f2, $2 }
' $*
```

range.sort:

```
# range.sort
#   Input/Output: string (tab) [b/e] (tab) number
#   Sort by $1 (string), $3 (number), then $2 (string)
sort '-t        ' +0 -1 +2n +1 -2 $*
```

range.collapse:

```
awk ' # range.collapse
#   Input:  string (tab) [b/e] (tab) number
#   Output: string (tab) num [(space) num]
function error(s) {
        print "range.collapse: " s " near pp " rlo "-" rhi | "cat 1>&2"
}
function printoldrange() {
        if (range == 1) { error("no %end for " term); rhi = "XXX" }
        if (NR > 1) print term, (rlo == rhi) ? rlo : (rlo " " rhi)
        rlo = rhi = $3  # bounds of current range
}

BEGIN         { FS = OFS = "\t" }
$1 != term    { printoldrange(); term = $1; range = 0 }
$2 == "e"     { if (range == 1) { range = 0; rhi = $3 }
                  else { printoldrange(); error("no %begin for " term);
                         rlo = "XXX"
                       }
                  next
                }
$3 <= rhi + 1 { rhi = $3}
$3 >  rhi + 1 { if (range == 0) printoldrange() }
$2 == "b"     { if (range == 1) error("multiple %begin for " term); range = 1 }
END           { if (NR == 1) NR = 2; printoldrange() }
' $*
```

This is the most subtle program; much of the complexity is a result of error checking. The simplified version shown below as `range.col.mini` removes the error checking.

range.col.mini:

```
awk ' # range.col.mini -- range.collapse with no error checking
function printoldrange() { # print existing range then initialize new range
        if (NR > 1) print term, (rlo == rhi) ? rlo : (rlo " " rhi)
        rlo = rhi = $3  # bounds of current range
}
BEGIN         { FS = OFS = "\t" }
$1 != term    { printoldrange(); term = $1; range = 0 } # new term
$2 == "e"     { range = 0; rhi = $3; next }             # end of range
$3 <= rhi + 1 { rhi = $3 }                              # continue existing
                                                        # range
$3 >  rhi + 1 { if (range == 0) printoldrange() }       # if explicit, print
                                                        # last range
$2 == "b"       {       range = 1 }                     # now in explicit
                                                        # range
END           { if (NR == 1) NR = 2; printoldrange() }  # print existing
                                                        # range
' $*                                                    # even on 1-line
                                                        # file
```

This program is not used by the indexing program, but it might help you understand `range.collapse`.

reroman:

```
awk ' # reroman
#   Input:  string (tab) arab1 [(space) arab2]
#   Output: string (tab) roman1 [-roman2]
BEGIN   { FS = OFS = "\t"
          # set a[1] = "i", a[2] = "ii", ...
          s =   "i ii iii iv v vi vii viii ix x"
          s = s " xi xii xiii xiv xv xvi xvii xviii xix xx"
          s = s " xxi xxii xxiii xxiv xxv xxvi xxvii xxviii xxix xxx"
          split(s, a, " ")
        }
$2 < 0  { n = split($2, b, " ")
          for (i = 1; i <= n; i++) {
                if (b[i] >= 0) continue
                j = 1000 + b[i]
                if (j in a) b[i] = a[j]
                else print "reroman: bad number: " $0  | "cat 1>&2"
          }
          $2 = b[1]
          if (n > 1) $2= b[1] " " b[2]
        }
        { sub(/ /, "-", $2); print }
' $*
```

num.collapse:

```
awk ' # num.collapse
#   Input:  string (tab) roman1 [-roman2]
#   Output: string (tab) numlist
BEGIN   { FS = OFS = "\t" }
$1 != p { p = $1
          if (NR > 1) printf "\n"
          printf "%s\t%s", $1, $2
          next
        }
        { printf " %s", $2 }
END     { if (NR > 0) printf "\n" }
' $*
```

The variable `p` is the previous value. The output uses space as a separator between numbers in the list.

rotate:

```
awk ' # rotate
#   Input:  string [%key sort key] (tab) numlist
#   Output: several rotations of string (tab) [key] (tab) numlist
BEGIN    { FS = OFS = "\t" }
/ %key / { i = index($1, " %key ")
           print substr($1, 1, i-1), substr($1, i+6), $2
           next
         }
         { print $1, "", $2
           i = 1
           while ((j = index(substr($1, i+1), " ")) > 0) {
                i += j
                printf("%s, %s\t\t%s\n",substr($1,i+1),substr($1,1,i-1),$2)
           }
         }
' $*
```

The tricky code in the `while` loop makes quite a difference in run time.

gen.key:

```
awk ' # gen.key
#   Input:  string (tab) [opt explicit key] (tab) numlist
#   Output: sort key (tab) string (tab) numlist
BEGIN { FS = OFS = "\t" }
$2 == "" { # generate key if none specified
        $2 = $1
        # Remove these troff commands:
        gsub(/\\f\(..|\\f.|\\s[+-][0-9]|\\s[0-9][0-9]?/, "", $2)
        # Def 1: keep blanks, letters, digits only
        #       gsub(/[^a-zA-Z0-9 ]+/, "", $2)
        # Def 2: remove index commands []{}, and % before literals
        atsign = 0
        if ($2 ~ /%/) {  # hide literals in @
                atsign = 1
                gsub(/%%/,  "@@0@@", $2)
                gsub(/%\[/, "@@1@@", $2)
                gsub(/%\]/, "@@2@@", $2)
                gsub(/%\{/, "@@3@@", $2)
                gsub(/%\}/, "@@4@@", $2)
                gsub(/%~/,  "@@5@@", $2)
        }
        gsub(/%e/, "\\", $2)
        gsub(/~/, " ", $2)
        gsub(/[%\[\]\{\}]/, "", $2)  # remove font-changing []{} and %, ~
        if (atsign) {  # replace literals
                gsub(/@@0@@/, "%", $2)
                gsub(/@@1@@/, "[", $2)
                gsub(/@@2@@/, "]", $2)
                gsub(/@@3@@/, "{", $2)
                gsub(/@@4@@/, "}", $2)
                gsub(/@@5@@/, "~", $2)
        }
        if ($2 ~ /^[^a-zA-Z]/) {  # force punctuation before a
                if ($2 ~ /^[0-9]/) $2 = " " $2
                else $2 = "  " $2
        }
}
        { print $2, $1, $3 }
' $*
```

Under Definition 1, the sort key consists of all alphanumeric characters in the string and this definition is commented out. Definition 2 is active; it tries to remove formatting commands.

final.sort:

```
# final.sort
#   Input/Output: sort key (tab) string (tab) numlist
#   Sort by $1 (string)
sort -f $*
```

The `-f` option folds upper and lower case together in comparisons.

format:

```
awk ' # format
#   Input:  sort key (tab) string (tab) numlist
#   Output: troff format, commands interpreted
BEGIN { FS = "\t"
        s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz "
        # set upper["a"] = "A"
        for (i = 1; i <= 27; i++) upper[substr(s,i+27,1)] = substr(s,i,1)
        # set lower["a"] =  lower["A"] ="a"
        for (i = 1; i <= 27; i++) {
                lower[substr(s,i,1)] = substr(s,i+27,1)
                lower[substr(s,i+27,1)] = substr(s,i+27,1)
        }
      }
      { # mark change between letters with .YY
        this = substr($1,1,1)
        if (!(this in lower)) lower[this] = " "
        this = lower[this]
        if (this != last && this != " ")
                print ".YY", this, upper[last=this]
        gsub(/ /, ", ", $3)      # commas between page numbers
        atsign = 0
        if ($2 ~ /%/) {
                atsign = 1
                gsub(/%%/,  "@@0@@", $2)
                gsub(/%\[/, "@@1@@", $2)
                gsub(/%\]/, "@@2@@", $2)
                gsub(/%\{/, "@@3@@", $2)
                gsub(/%\}/, "@@4@@", $2)
                gsub(/%~/,  "@@5@@", $2)
        }
        gsub(/%e/, "\\e", $2) # %e -> \e
        gsub(/~/, " ", $2)    # tildes go away at last
        if (gsub(/\[/, "\\\\&\\f(CW", $2))
                gsub(/\]/, "\\fP",   $2)
        if (gsub(/\{/, "\\f2",    $2))
                gsub(/\}/, "\\fP",   $2)
        if (atsign) {
                gsub(/%/, "", $2)
                gsub(/@@0@@/, "%", $2)
                gsub(/@@1@@/, "[", $2)
                gsub(/@@2@@/, "]", $2)
                gsub(/@@3@@/, "{", $2)
                gsub(/@@4@@/, "}", $2)
                gsub(/@@5@@/, "~", $2)
        }
        print ".XX"; printf "\\&%s  %s\n", $2, $3
      }
' $*
```

There is no good way to convert cases in *awk*.