Computing Science Technical Report No. 128

Tools for Printing Indexes

Jon L. Bentley
Brian W. Kernighan
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

October, 1986

Tools for Printing Indexes

Jon L. Bentley
Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes a set of programs for processing and printing the index for a book or a manual. The input is a set of lines containing index terms and page numbers. (Disclaimer: these programs do *not* help with the original creation of index terms!) 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.) and with roman numerals.

The implementation uses an unusual software style: a long pipeline of short *awk* programs. This structure makes the programs easy to adapt or augment to meet the special requirements that arise in many indexes. The programs were intended to be used with *troff*, but can be used with TEX or *monk*[1] with minor changes.

October, 1986

Tools for Printing Indexes

Jon L. Bentley
Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. 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 half a dozen books and as many programmer manuals, and have never found a substitute for a lot of thought.

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

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. And if we hadn't foreseen your particular problem, you might find it hard to modify the code.

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 exotic features (formatting options, glossary definitions, hierarchies, 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 next two sections describe how to use the package in its current form; nonprogrammers will probably not want to read past Section 3. Sections 4 and 5 discuss implementation and modification, and Section 6 contains remarks on the style of programming.

2. Typical Use

The first step 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. A more satisfactory way, however, is 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 *troff*, a call of the macro `.ix` can be used:

```
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 ...
```

The `.ix` macro is not part of the standard macro packages like `-mm` or `-ms`; fortunately its definition is short:†

```
.de ix
.tm ix: \\$1 \\$2 \\$3 \\$4 \\$5 \\$6 \\$7 \\$8 \\$9   \\n%
..
```

This causes the (up to nine) words of the index term, a tab, and the current page number to be written on *troff*'s standard error output. The string ''`ix:`'' is added to the front so index terms can later be separated from anything else that might have been printed on the error output. [Warning: some of *troff*'s special characters don't come through `.tm` commands at all and others come through in strange ways.]

The index output is captured in a file, say `ix.raw`, by redirecting file 2 (`stderr`):

```
troff -ms ... >t.out 2>ix.raw
```

(Typical documents also use *refer*, *pic*, *tbl*, *eqn*, etc., in a pipeline.) The raw output is processed by the program `make.index`:

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

Each line written to `index.body` is preceded by a call to macro `.XX`, which can be used later to format it. In addition, a call to the `.YY` macro is generated before each new letter of the alphabet; after *azure* and before *Babbage*, `.YY` is called with the two arguments `b` and `B`.

The complete index is generated by a later *troff* run:

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

The standard file `index.head` contains default definitions of the `.XX` and `.YY` macros, plus other commands to set multiple columns, etc. This file is set up to produce a three-column index on a normal six-inch page width, using the `.MC` macro of the `-ms` macro package. If you use some other macro package, you will have to change this file.

The books [2] and [3] illustrate the style of index produced by this program (although those books used earlier versions).

---

† The version of the `.ix` macro in the appendix is slightly more complicated so as to handle index terms contained within diverted text like floating keeps.

3. Input of Index Terms

A simple language is used within arguments to the `.ix` macro to control fonts, permutations of phrases, and order of sorting. The general form of an indexing entry is

```
.ix this is a phrase
```

which, if it occurs on page 97, will be written in the `ix.raw` file as

```
this is a phrase    (tab)97
```

This will subsequently be converted into four entries by rotating the phrase around each blank:

```
this is a phrase    97
phrase, this is a   97
a phrase, this is   97
is a phrase, this   97
```

The character `~` is translated to a blank, so it may be used to control the automatic rotation:

```
.ix this~is~a phrase
```

will eventually produce

```
this is a phrase    97
phrase, this is a   97
```

If there are multiple occurrences of an indexing phrase on adjacent pages, they will be collected and merged to appear as, for example, 97, 99, 108-110.

Page numbers in lower case roman numerals (i, ii, iii, ...) are sorted before arabic page numbers.

Within the words of an index phrase, the following special constructs are recognized:

`~`	will print as blank
`[...]`	will print ... in `CW font`
`{...}`	will print ... in *italics*
`_[, _]`	literal `[, ]`
`_{, _}`	literal `{, }`
`%`	explicit sort key follows
`_%`	literal `%`
`istart`	start a range of page numbers
`iend`	end a range

For example,

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

produces

> `%d` specification, `printf`
> `[^...]` regular expression
> command, `pr` *−n*
> `pr` *−n* command
> `printf %d` specification
> regular expression, `[^...]`
> specification, `printf %d`

Only the specific nesting `[{}]` currently works for font changes. There is also no way to print a literal `~`. Proper handling of `_~` is left as an exercise; see Section 5.

You may want to produce an entry like ‘‘*large subject 19-35*’’, to indicate a range of pages. Two special `.ix` entries can be used to define the range:

```
.ix istart large subject    .. on the first page
.ix iend large subject      .. on the last page
```

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`, size changes like `\s8` and `\s-3`, and other miscellany.

If the sort order isn't what you want, you may force a sort key by using the `%...` construct:

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

as in

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

(Notice that no space precedes the `%`.)

If you use complicated keys, you will have to study the program *gen.key* to use explicit keys effectively. For instance, that program controls grouping by prepending a single space to a string that starts with a number and prepending two spaces to a string that starts with punctuation.

4. Principles of Operation

The indexer consists of a host of small *awk* programs, intentionally kept separate for easy modification. For example, roman numeral page numbers are processed by *deroman* and *reroman.* These programs can only count up to *xxx*, however, so you must make a simple change to them if you want to count beyond 30. On the other hand, if you don't have any roman-numeral pages, you don't need *deroman* and *reroman* at all.

The basic strategy is to sort once to bring together all occurrences of identical index terms so as to combine their page numbers. 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 `%` command allows you to override the default sort key.

The shell file *make.index* controls the process:

```
make.index ix.raw ... >index.out
```

The specific programs are, in order,

`doclean`	strip excess spaces before the tabs, remove non-`ix:` lines
`deroman`	map roman numerals to arabic
`range.prep`	prepare to sort (handle `istart`/`iend`)
`range.sort`	sort by string then page number
`range.collapse`	resolve `istart`/`iend` 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.

A few implementation details may prove useful. The *awk* programs rely on features in the *awk* interpreter released in mid-1985 [4]. If your *awk* gets a syntax error on this program

```
awk '{ gsub(/A-Z/, "") }'
```

to remove capital letters, you must install an up-to-date version. The ‘‘pipeline’’ actually uses temporary files *foo0*, *foo1*, ... to connect the stages (some systems have a small limit on the number of filters in a pipeline; the intermediate files are also useful for debugging), and deletes the files at the end. On a VAX-11/750, `make.index` takes a few minutes for a medium-sized book (500 distinct entries, 1500 total entries).

5. Bells and Whistles

Our programs produce a basic index. If you want additional features, you will have to build them yourself by adding to or adapting our tools.

To illustrate the process, we'll consider a simple addition: ''*see*'' references of the form

secondary *see* primary

(The examples in the introduction also illustrate a *see also* reference to follow a list of page numbers; we'll leave that as an exercise.) The *see* references are contained in the file `see.terms` in the format

secondary(tab)primary

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 -m junk.see junk.regular | format >index.body
rm junk.regular junk.see
```

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 ' 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 simple construction assumes that the terms contain no formatting commands; if they do, you must pipe them through `gen.key` as well.

If you desire a minor cosmetic change to the style of the index, you will probably have to alter the *troff* commands in the header file `index.head`. Our tastes run to ragged-right columns in the short lines of an index; if you prefer aligned columns, remove the `.na` line (and be prepared for some funny line filling). The `.YY` macro in that file places the new letter of the alphabet between letter breaks, to show how that might be done. We prefer just to leave a small space, so we define `.YY` as

```
.de YY    \" header between letters of the alphabet
.sp 1.5
..
```

For more radical stylistic changes, you may have to modify the `format` program as well.

The most obvious missing piece in our suite 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

If you really want a hierarchical index, we recommend 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.) Your implementation can probably get by with changing only the `format` program and the `index.head` file; you will probably need a new macro `.ZZ` for secondary terms.

Some people don't want to collapse runs of adjacent page numbers. They feel that the list ''5, 6, 7'' implies three scattered references on those pages, while the sequence ''5-7'' implies a lengthy discussion. If you are in that camp, you must provide the ranges explicitly using `istart` and `iend` commands, then modify the program `range.collapse` to avoid merging adjacent page numbers.

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 *monk* or 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 [5] is essentially identical to our `.ix` macro, while *monk* uses `|index(`*term*`)`. You must then modify `doclean` to sweep up any loose ends and `format` to produce output of the right form; the rest of the pipe is unchanged. Your last job is to incorporate the resulting output into the document, using a mechanism like `index.head`.

We have also used the `.ix` macro to generate text and page numbers for tables of contents. A macro for producing section heads, 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 *sed* or even (as a last resort) a text editor to fix up things that just don't work right.

6. 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 [6]. As capabilities were added over the years, the programs degraded into write-only code, penetrable only with substantial effort.

Our motivation to build the current suite of *awk* programs was preparing the index to Reference [2]. That index was substantially different from those processed by the existing programs: it had no font changes (which 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 indexing suite, 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 final version of the C program, the initial *awk* program, and the final *awk* program are summarized in Table 1.

PROGRAM	C PROTOTYPE	AWK PROTOTYPE	AWK PRODUCTION
`doclean`	3 sed		11
`deroman`	7 sed	7	17
`range.prep`			9
`range.sort`			4 sh
`range.collapse`			43
`reroman`	4 sed	10	22
`num.collapse`	49 C	4	12
`rotate`	60 C	6	21
`gen.key`	18 sed		34
`final.sort`	1 sh	1 sh	4 sh
`format`	30 sed	10	47
Total Lines	172	37	224

Table 1. Lines of Source Code

Table 1 has been massaged to compare incomparables. The C suite, 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. (More details on the C programs shortly.) The C suite 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.

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* coding tricks reduced the run time of some individual filters by an order of magnitude.

Readability. Although this may stretch the imagination of some readers, the first version was much less readable than the code presented in the appendix. (Fifty of the 224 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 a very 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 one monolithic program for producing an index for *troff* books; it is 350 lines of prototype-quality C (and makes use of several system utilities). 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].)

For ease of implementation, the *awk* language is a substantial improvement over C: *awk* is much better suited 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. (Any automatic process can create insidious errors if its output is accepted blindly. For one instance, the index entry for `kill -3` in the index of Reference [1] was rendered as `kill iv`; the reader might enjoy trying to infer the combination of circumstances that caused this gaffe.)

Performance does suffer in the *awk* version. The index to Reference [1] (1770 input lines) takes 109 seconds with the old C version and 234 seconds with the new one on a VAX-11/750. This factor of two is

acceptable for a program that is run only occasionally. The run times of the new programs and the size of the data flowing between them are presented in this profile:

```
lines  words   chars
                      times      program
1770    6606   53483
                18.1u 1.2s 21r       doclean
1681    4276   33568
                 8.1u 1.2s 10r       deroman
1681    4276   33688
                12.4u 1.5s 16r       range.prep
1681    4276   35369
                19.6u 1.4s 23r       range.sort
1681    4276   35369
                26.5u 1.8s 33r       range.collapse
1639    4203   32978
                 9.0u 1.3s 11r       reroman
1639    4174   32858
                11.3u 1.3s 15r       num.collapse
1161    3464   25680
                21.8u 1.6s 25r       rotate
1825    7141   45142
                27.4u 1.9s 32r       gen.key
1825   11722   74292
                16.2u 1.8s 26r       final.sort
1825   11722   74292
                43.8u 2.6s 51r       format
3675    9029   66443
```

Here is a similar profile of the C prototype:

```
lines  words   chars
                      times      program
1770    6606   53483
                 5.2u 0.7s  6r       sed -f doclean_deroman
1770    4836   37181
                 3.7u 0.5s  5r       rotate
3028   10296   76409
                22.6u 1.5s 25r       sed -f gen.key
2595   10836   65629
                41.0u 2.9s 58r       final.sort
2595   10836   65629
                 3.5u 0.6s  7r       num.collapse
1793    8664   51227
                23.8u 2.9s 47r       sed -f format_reroman
3586    8855   65303
```

(The file sizes are close but not equal: we used one input file as input to two programs with slightly different functionality.)

Acknowledgments

We are grateful to Al Aho, Ted Kowalski, Doug McIlroy, Ravi Sethi, Chris Van Wyk and Pamela Zave 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. Murrel, S. L. and Kowalski, T. J., *Typing Documents on the UNIX System: Using Monk 0.3*, Bell Laboratories internal memorandum (December 10, 1985).
2. Brian W. Kernighan and Rob Pike, *The Unix Programming Environment,* Prentice-Hall (1984).
3. Jon L. Bentley, *Programming Pearls,* Addison-Wesley (1986).
4. Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, ‘‘AWK: A Pattern Scanning and Processing Language (User’s Manual),’’ CSTR 118 (June 1985).
5. Leslie Lamport, *LATEX: A Document Preparation System,* Addison-Wesley (1986).
6. Brian W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley (1976).
7. David L. Parnas, ‘‘On the criteria to be used in decomposing systems into modules,’’ *Communications of the ACM* **15**(12), pp. 1053-1058 (December 1972).

Appendix: The Programs

This appendix lists the programs verbatim in the order in which they are used; the programs are available from the authors. But first, a summary of the commands. Fields in square brackets [] are optional.

```
doclean
     Input:  string (blanks and tab) number
     Output: string (tab) number
deroman
     Input:  string (tab) arab or roman
     Output: string (tab) arab
     Roman numeral n is replaced by arab n-1000 (i.e., iii -> -997)
range.prep
     Input:  [istart/iend] string (tab) number
     Output: string (tab) [b/e] (tab) number
range.sort
     Sort by $1 (string), $2 (string), then $3 (number)
range.collapse
     Input:  string (tab) [b/e] (tab) number
     Output: string (tab) num [(space) num]
reroman
     Input:  string (tab) arab1 [(space) arab2]
     Output: string (tab) roman1 [-roman2]
num.collapse
     Input:  string (tab) roman1 [-roman2]
     Output: string (tab) numlist
rotate
     Input:  string [%optional sort key] (tab) numlist
     Output: rotations of string (1/line) (tab) [key] (tab) numlist
gen.key
     Input:  string (tab) [opt explicit key] (tab) numlist
     Output: sort key (tab) string (tab) numlist
final.sort
     Sort by $1 (string) folded to lower case
format
     Input:  sort key (tab) string (tab) numlist
     Output: troff format, commands interpreted
```

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.
.\" if you use -mm, you will want to change .SH
.\" to some form of .HU "heading", and .LP to .P 0.
.\" the number registers PS and VS are different too.
.
.
.pn 999         \" page number for first page; set it to taste
.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:

```
doclean $* >foo1
deroman foo1 >foo2
range.prep foo2 >foo3
range.sort foo3 >foo4
range.collapse foo4 >foo5
reroman foo5 >foo6
num.collapse foo6 >foo7
rotate foo7 >foo8
gen.key foo8 >foo9
final.sort foo9 >junk.regular

see.prep see.terms | gen.key | final.sort >junk.see
sort -mfd junk.see junk.regular | format >junk.all
cat index.head junk.all

# rm foo* junk*
```

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 } # CONT. marks table of contents stuff

          { 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 is used 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)

#  this version doesn't work with page numbers like 4-56

sort -u '-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) {
          if (rlo == rhi)
               print term, rlo
          else
               print term, (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 program would be much shorter without error checking.

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]
        }
        { print }
' $*
```

num.collapse:

```
awk ' # num.collapse
#   Input:  string (tab) roman1 [-roman2]
#   Output: string (tab) numlist

BEGIN     { FS = OFS = "\t" }

     { sub(/ /, "\\(en", $2) }  # use - if there is no en dash

$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(with commas)

BEGIN      { FS = OFS = "\t" }

      { # convert page page page into page, page, page
        # ought to be in num.collapse
          gsub(/ /, ", ", $2)      # commas between page numbers
      }

/ %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 last `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
     # quote character is %, space character is ~
     quoted = 0
     if ($2 ~ /%/) {  # hide literals in Q
          quoted = 1
          gsub(/%%/,  "QQ0QQ", $2)
          gsub(/%\[/, "QQ1QQ", $2)
          gsub(/%\]/, "QQ2QQ", $2)
          gsub(/%\{/, "QQ3QQ", $2)
          gsub(/%\}/, "QQ4QQ", $2)
          gsub(/%~/,  "QQ5QQ", $2)
     }
     gsub(/%e/, "\\", $2)          # troff escape
     gsub(/~/, " ", $2)
     gsub(/[%\[\]\{\}]/, "", $2)   # remove font-changing []{} and %, ~
     if (quoted) {  # replace literals
          gsub(/QQ0QQ/, "%", $2)
          gsub(/QQ1QQ/, "[", $2)
          gsub(/QQ2QQ/, "]", $2)
          gsub(/QQ3QQ/, "{", $2)
          gsub(/QQ4QQ/, "}", $2)
          gsub(/QQ5QQ/, "~", $2)
     }
     if ($2 ~ /^[^a-zA-Z]+$/) # pure punctuation goes first
          $2 = "  " $2
     else if ($2 ~ /^[0-9]/)       # leading digits come next
          $2 = " " $2
                          # otherwise whatever final.sort does
}

     { print $2, $1, $3 }
' $*
```

Under Definition 1, the sort key consists of all alphanumeric characters in the string; that 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 -fd $*
sort -t'  ' +0fd -1 -t' ' +0f -1 $*
```

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
     # find first non-punctuation char
     for (i = 1; (c = substr($1,i,1)) != ""; i++)
          if (c ~ /[a-zA-Z0-9 ]/)
               break
     this = c
     if (!(this in lower)) lower[this] = " "
     this = lower[this]
     if (this != last && this != " ")
          print ".YY", this, upper[last=this]
     quoted = 0

  # interpret font change language

     $0 = $2 "  " $3     # discard sort key, leave term .. numlist

     if ($0 ~ /%/) {
          quoted = 1
          gsub(/%%/,  "QQ0QQ", $0)
          gsub(/%\[/, "QQ1QQ", $0)
          gsub(/%\]/, "QQ2QQ", $0)
          gsub(/%\{/, "QQ3QQ", $0)
          gsub(/%\}/, "QQ4QQ", $0)
          gsub(/%~/,  "QQ5QQ", $0)
     }
     gsub(/%e/, "\\e", $0) # %e -> \e
     gsub(/~/, " ", $0)    # unpaddable spaces go away at last
     if (gsub(/\[/, "\\\\&\\f(CW", $0))
          gsub(/\]/, "\\fP",   $0)
     if (gsub(/\{/, "\\f2",   $0))
          gsub(/\}/, "\\fP",   $0)
     if (quoted) {
          gsub(/%/, "", $0)
          gsub(/QQ0QQ/, "%", $0)
          gsub(/QQ1QQ/, "[", $0)
          gsub(/QQ2QQ/, "]", $0)
          gsub(/QQ3QQ/, "{", $0)
          gsub(/QQ4QQ/, "}", $0)
          gsub(/QQ5QQ/, "~", $0)
     }
     print ".XX"; printf "\\&%s\n", $0
}
' $*
```

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