Some Applications of Inverted Indexes on the UNIX System

M. E. Lesk

ABSTRACT

I. Some Applications of Inverted Indexes – Overview

This memorandum describes a set of programs which make inverted indexes to UNIX* text files, and their application to retrieving and formatting citations for documents prepared using *troff.*

The indexing and searching programs make keyword indexes to volumes of material too large for linear searching. Searches for combinations of single words can be performed quickly. The programs for general searching are divided into two phases. The first makes an index from the original data; the second searches the index and retrieves items. Both of these phases are further divided into two parts to separate the data-dependent and algorithm dependent code.

The major current application of these programs is the *troff* preprocessor *refer.* A list of 4300 references is maintained on line, containing primarily papers written and cited by local authors. Whenever one of these references is required in a paper, a few words from the title or author list will retrieve it, and the user need not bother to re-enter the exact citation. Alternatively, authors can use their own lists of papers.

This memorandum is of interest to those who are interested in facilities for searching large but relatively unchanging text files on the UNIX system, and those who are interested in handling bibliographic citations with UNIX *troff.*

II. Updating Publication Lists

This section is a brief note describing the auxiliary programs for managing the updating processing. It is written to aid clerical users in maintaining lists of references. Primarily, the programs described permit a large amount of individual control over the content of publication lists while retaining the usefulness of the files to other users.

III. Manual Pages

This section contains the pages from the UNIX programmer's manual dealing with these commands. It is useful for reference.

---

* UNIX is a Trademark of Bell Laboratories.

1. Introduction.

The UNIX† system has many utilities (e.g. *grep, awk, lex, egrep, fgrep, ...*) to search through files of text, but most of them are based on a linear scan through the entire file, using some deterministic automaton. This memorandum discusses a program which uses inverted indexes %A D. Knuth %T The Art of Computer Programming: Vol. 3, Sorting and Searching %I Addison-Wesley %C Reading, Mass. %D 1977 %O See section 6.5. and can thus be used on much larger data bases.

As with any indexing system, of course, there are some disadvantages; once an index is made, the files that have been indexed can not be changed without remaking the index. Thus applications are restricted to those making many searches of relatively stable data. Furthermore, these programs depend on hashing, and can only search for exact matches of whole keywords. It is not possible to look for arithmetic or logical expressions (e.g. ‘‘date greater than 1970’’) or for regular expression searching such as that in *lex.* lex lesk cstr

Currently there are two uses of this software, the *refer* preprocessor to format references, and the *lookall* command to search through all text files on the UNIX system.

The remaining sections of this memorandum discuss the searching programs and their uses. Section 2 explains the operation of the searching algorithm and describes the data collected for use with the *lookall* command. The more important application, *refer* has a user’s description in section 3. Section 4 goes into more detail on reference files for the benefit of those who wish to add references to data bases or write new *troff* macros for use with *refer.* The options to make *refer* collect identical citations, or otherwise relocate and adjust references, are described in section 5. The UNIX manual sections for *refer, lookall,* and associated commands are attached as appendices.

2. Searching.

The indexing and searching process is divided into two phases, each made of two parts. These are shown below.

† UNIX is a trademark of Bell Laboratories.

A. Construct the index.

(1) Find keys — turn the input files into a sequence of tags and keys, where each tag identifies a distinct item in the input and the keys for each such item are the strings under which it is to be indexed.

(2) Hash and sort — prepare a set of inverted indexes from which, given a set of keys, the appropriate item tags can be found quickly.

B. Retrieve an item in response to a query.

(3) Search — Given some keys, look through the files prepared by the hashing and sorting facility and derive the appropriate tags.

(4) Deliver — Given the tags, find the original items. This completes the searching process.

The first phase, making the index, is presumably done relatively infrequently. It should, of course, be done whenever the data being indexed change. In contrast, the second phase, retrieving items, is presumably done often, and must be rapid.

An effort is made to separate code which depends on the data being handled from code which depends on the searching procedure. The search algorithm is involved only in programs (2) and (3), while knowledge of the actual data files is needed only by programs (1) and (4). Thus it is easy to adapt to different data files or different search algorithms.

To start with, it is necessary to have some way of selecting or generating keys from input files. For dealing with files that are basically English, we have a key-making program which automatically selects words and passes them to the hashing and sorting program (step 2). The format used has one line for each input item, arranged as follows:

name:start,length (tab) key1 key2 key3 ...

where *name* is the file name, *start* is the starting byte number, and *length* is the number of bytes in the entry.

These lines are the only input used to make the index. The first field (the file name, byte position, and byte count) is the tag of the item and can be used to retrieve it quickly. Normally, an item is either a whole file or a section of a file delimited by blank lines. After the tab, the second field contains the keys. The keys, if selected by the automatic program, are any alphanumeric strings which are not among the 100 most frequent words in English and which are not entirely numeric (except for four-digit numbers beginning 19, which are accepted as dates). Keys are truncated to six characters and converted to lower case. Some selection is needed if the original items are very large. We normally just take the first *n* keys, with *n* less than 100 or so; this replaces any attempt at intelligent selection. One file in our system is a complete English dictionary; it would presumably be retrieved for all queries.

To generate an inverted index to the list of record tags and keys, the keys are hashed and sorted to produce an index. What is wanted, ideally, is a series of lists showing the tags associated with each key. To condense this, what is actually produced is a list showing the tags associated with each hash code, and thus with some set of keys. To speed up access and further save space, a set of three or possibly four files is produced. These files are:

center; c c lI l. File Contents entry Pointers to posting file for each hash code posting Lists of tag pointers for each hash code tag Tags for each item key Keys for each item (optional)

The posting file comprises the real data: it contains a sequence of lists of items posted under each hash code. To speed up searching, the entry file is an array of pointers into the posting file, one per potential hash code. Furthermore, the items in the lists in the posting file are not referred to by their complete tag, but just by an address in the tag file, which gives the complete tags. The key file is optional and contains a copy of the keys used in the indexing.

The searching process starts with a query, containing several keys. The goal is to obtain all items which were indexed under these keys. The query keys are hashed, and the pointers in the entry file used to access the lists in the posting file. These lists are addresses in the tag file of documents posted under the hash codes derived from the query. The common items from all lists are determined; this must include the items indexed by every key, but may also contain some items which are false drops, since items referenced by the correct hash codes need not actually have contained the correct keys. Normally, if there are several keys in the query, there are not likely to be many false drops in the final combined list even though each hash code is somewhat ambiguous. The actual tags are then obtained from the tag file, and to guard against the possibility that an item has false-dropped on some hash code in the query, the original items are normally obtained from the delivery program (4) and the query keys checked against them by string comparison.

Usually, therefore, the check for bad drops is made against the original file. However, if the key derivation procedure is complex, it may be preferable to check against the keys fed to program (2). In this case the optional key file which contains the keys associated with each item is generated, and the item tag is supplemented by a string

;start,length

which indicates the starting byte number in the key file and the length of the string of keys for each item. This file is not usually necessary with the present key-selection program, since the keys always appear in the original document.

There is also an option (**-C***n*) for coordination level searching. This retrieves items which match all but *n* of the query keys. The items are retrieved in the order of the number of keys that they match. Of course, *n* must be less than the number of query keys (nothing is retrieved unless it matches at least one key).

As an example, consider one set of 4377 references, comprising 660,000 bytes. This included 51,000 keys, of which 5,900 were distinct keys. The hash table is kept full to save space (at the expense of time); 995 of 997 possible hash codes were used. The total set of index files (no key file) included 171,000 bytes, about 26% of the original file size. It took 8 minutes of processor time to hash, sort, and write the index. To search for a single query with the resulting index took 1.9 seconds of processor time, while to find the same paper with a sequential linear search using *grep* (reading all of the tags and keys) took 12.3 seconds of processor time.

We have also used this software to index all of the English stored on our UNIX system. This is the index searched by the *lookall* command. On a typical day there were 29,000 files in our user file system, containing about 152,000,000 bytes. Of these 5,300 files, containing 32,000,000 bytes (about 21%) were English text. The total number of ‘words’ (determined mechanically) was 5,100,000. Of these 227,000 were selected as keys; 19,000 were distinct, hashing to 4,900 (of 5,000 possible) different hash codes. The resulting inverted file indexes used 845,000 bytes, or about 2.6% of the size of the original files. The particularly small indexes are caused by the fact that keys are taken from only the first 50 non-common words of some very long input files.

Even this large *lookall* index can be searched quickly. For example, to find this document by looking for the keys ‘‘lesk inverted indexes’’ required 1.7 seconds of processor time and system time. By comparison, just to search the 800,000 byte dictionary (smaller than even the inverted indexes, let alone the 27,000,000 bytes of text files) with *grep* takes 29 seconds of processor time. The *lookall* program is thus useful when looking for a document which you believe is stored on-line, but do not know where. For example, many memos from our center are in the file system, but it is often difficult to guess where a particular memo might be (it might have several authors, each with many directories, and have been worked on by a secretary with yet more directories). Instructions for the use of the *lookall* command are given in the manual section, shown in the appendix to this memorandum.

The only indexes maintained routinely are those of publication lists and all English files. To make other indexes, the programs for making keys, sorting them, searching the indexes, and delivering answers must be used. Since they are usually invoked as parts of higher-level commands, they are not in the default command directory, but are available to any user in the directory */usr/lib/refer*. Three programs are of interest: *mkey*, which isolates keys from input files; *inv*, which makes an index from a set of keys; and *hunt*, which searches the index and delivers the items. Note that the two parts of the retrieval phase are combined into one program, to avoid the excessive system work and delay which would result from running these as separate processes.

These three commands have a large number of options to adapt to different kinds of input. The user not interested in the detailed description that now follows may skip to section 3, which describes the *refer* program, a packaged-up version of these tools specifically oriented towards formatting references.

Make Keys. The program *mkey* is the key-making program corresponding to step (1) in phase A. Normally, it reads its input from the file names given as arguments, and if there are no arguments it reads from the standard input. It assumes that blank lines in the input delimit separate items, for each of which a different line of keys should be generated. The lines of keys are written on the standard output. Keys are any alphanumeric string in the input not among the most frequent words in English and not entirely numeric (except that all-numeric strings are acceptable if they are between 1900 and 1999). In the output, keys are translated to lower case, and truncated to six characters in length; any associated punctuation is removed. The following flag arguments are recognized by *mkey:*

center; lB lw(4i). −c *name T{ Name of file of common words; default is /usr/lib/eign. T} −f name T{ Read a list of files from name and take each as an input argument. T} −i chars T{ Ignore all lines which begin with '%' followed by any character in chars . T} −kn T{ Use at most n keys per input item. T} −ln T{ Ignore items shorter than n letters long. T} −nm T{ Ignore as a key any word in the first m words of the list of common English words. The default is 100. T} −s T{ Remove the labels (file:start,length) from the output; just give the keys. Used when searching rather than indexing. T} −w T{ Each whole file is a separate item; blank lines in files are irrelevant. T}*

The normal arguments for indexing references are the defaults, which are *−c /usr/lib/eign* , *−n100* , and *−l3* . For searching, the *−s* option is also needed. When the big *lookall* index of all English files is run, the options are *−w* , *−k50* , and *−f (filelist)* . When running on textual input, the *mkey* program processes about 1000 English words per processor second. Unless the *−k* option is used (and the input files are long enough for it to take effect) the output of *mkey* is comparable in size to its input.

Hash and invert. The *inv* program computes the hash codes and writes the inverted files. It reads the output of *mkey* and writes the set of files described earlier in this section. It expects one argument, which is used as the base name for the three (or four) files to be written. Assuming an argument of *Index* (the default) the entry file is named *Index.ia* , the posting file *Index.ib* , the tag file *Index.ic* , and the key file (if present) *Index.id* . The *inv* program recognizes the following options:

center; lB lw(4i). −a T{ Append the new keys to a previous set of inverted files, making new files if there is no old set using the same base name. T} −d T{ Write the optional key file. This is needed when you can not check for false drops by looking for the keys in the original inputs, i.e. when the key derivation procedure is complicated and the output keys are not words from the input files. T} −h*n T{ The hash table size is n (default 997); n should be prime. Making n* bigger saves search time and spends disk space. T} −i[u] *name T{ Take input from file name , instead of the standard input; if* **u** *is present name is*

-- --

unlinked when the sort is started. Using this option permits the sort scratch space to overlap the disk space used for input keys. T} –n T{ Make a completely new set of inverted files, ignoring previous files. T} –p T{ Pipe into the sort program, rather than writing a temporary input file. This saves disk space and spends processor time. T} –v T{ Verbose mode; print a summary of the number of keys which finished indexing. T}

About half the time used in *inv* is in the contained sort. Assuming the sort is roughly linear, however, a guess at the total timing for *inv* is 250 keys per second. The space used is usually of more importance: the entry file uses four bytes per possible hash (note the **–h** option), and the tag file around 15-20 bytes per item indexed. Roughly, the posting file contains one item for each key instance and one item for each possible hash code; the items are two bytes long if the tag file is less than 65336 bytes long, and the items are four bytes wide if the tag file is greater than 65536 bytes long. Note that to minimize storage, the hash tables should be over-full; for most of the files indexed in this way, there is no other real choice, since the *entry* file must fit in memory.

Searching and Retrieving. The *hunt* program retrieves items from an index. It combines, as mentioned above, the two parts of phase (B): search and delivery. The reason why it is efficient to combine delivery and search is partly to avoid starting unnecessary processes, and partly because the delivery operation must be a part of the search operation in any case. Because of the hashing, the search part takes place in two stages: first items are retrieved which have the right hash codes associated with them, and then the actual items are inspected to determine false drops, i.e. to determine if anything with the right hash codes doesn't really have the right keys. Since the original item is retrieved to check on false drops, it is efficient to present it immediately, rather than only giving the tag as output and later retrieving the item again. If there were a separate key file, this argument would not apply, but separate key files are not common.

Input to *hunt* is taken from the standard input, one query per line. Each query should be in *mkey –s* output format; all lower case, no punctuation. The *hunt* program takes one argument which specifies the base name of the index files to be searched. Only one set of index files can be searched at a time, although many text files may be indexed as a group, of course. If one of the text files has been changed since the index, that file is searched with *fgrep;* this may occasionally slow down the searching, and care should be taken to avoid having many out of date files. The following option arguments are recognized by *hunt:*

center; lB lw(4i). −a T{ Give all output; ignore checking for false drops. T} −C*n T{ Coordination level n; retrieve items with not more than n terms of the input missing; default C0, implying that each search term must be in the output items. T} −F[ynd*] **T{ ''−Fy'' gives the text of all the items found; ''−Fn'' suppresses them. ''−F***d* '' where *d* is an integer gives the text of the first *d* items. The default is −*Fy.* T} −g T{ Do not use *fgrep* to search files changed since the index was made; print an error comment instead. T} −i *string T{ Take string as input, instead of reading the standard input. T} −l n T{ The maximum length of internal lists of candidate items is n; default 1000. T} −o string T{ Put text output (''−Fy'') in string; of use only when invoked from another program. T} −p T{ Print hash code frequencies; mostly for use in optimizing hash table sizes. T} −T[ynd*] **T{ ''−Ty'' gives the tags of the items found; ''−Tn'' suppresses them. ''−T***d* '' where *d* is an integer gives the first *d* tags. The default is −*Tn*. T} −t *string T{ Put tag output (''−Ty'') in string; of use only when invoked from another program. T}*

The timing of *hunt* is complex. Normally the hash table is overfull, so that there will be many false drops on any single term; but a multi-term query will have few false drops on all terms. Thus if a query is underspecified (one search term) many potential items will be examined and discarded as false drops, wasting time. If the query is overspecified (a dozen search terms) many keys will be examined only to verify that the single item under consideration has that key posted. The variation of search time with number of keys is shown in the table below. Queries of varying length were constructed to retrieve a particular document from the file of references. In the sequence to the left, search terms were chosen so as to select the desired paper as quickly as possible. In the sequence on the right, terms were chosen inefficiently, so that the query did not uniquely select the desired document until four keys had been used. The same document was the target in each case, and the final set of eight keys are also identical; the differences at five, six and seven keys are produced by measurement error, not by the slightly different key lists.

center; c s s s5 | c s s s cp8 cp8 cp8 cp8 | cp8 cp8 cp8 cp8 cp8 cp8 cp8 cp8 | cp8 cp8 cp8 cp8 n n n n | n n n n . Efficient Keys Inefficient Keys No. keys Total drops Retrieved Search time No. keys Total drops Retrieved Search time (incl.

false)	Documents	(seconds)				(incl.					false)	Documents			(seconds)
1	15	3	1.27	1	68	55	5.96	2	1	1	0.11	2	29	29	2.72
3	1	1	0.14	3	8	8	0.95	4	1	1	0.17	4	1	1	0.18
5	1	1	0.19	5	1	1	0.21	6	1	1	0.23	6	1	1	0.22
7	1	1	0.27	7	1	1	0.26	8	1	1	0.29	8	1	1	0.29

As would be expected, the optimal search is achieved when the query just specifies the answer; however, overspecification is quite cheap. Roughly, the time required by *hunt* can be approximated as 30 milliseconds per search key plus 75 milliseconds per dropped document (whether it is a false drop or a real answer). In general, overspecification can be recommended; it protects the user against additions to the data base which turn previously uniquely-answered queries into ambiguous queries.

The careful reader will have noted an enormous discrepancy between these times and the earlier quoted time of around 1.9 seconds for a search. The times here are purely for the search and retrieval: they are measured by running many searches through a single invocation of the *hunt* program alone. The normal retrieval operation involves using the shell to set up a pipeline through *mkey* to *hunt* and starting both processes; this adds a fixed overhead of about 1.7 seconds of processor time to any single search. Furthermore, remember that all these times are processor times: on a typical morning on our PDP 11/70 system, with about one dozen people logged on, to obtain 1 second of processor time for the search program took between 2 and 12 seconds of real time, with a median of 3.9 seconds and a mean of 4.8 seconds. Thus, although the work involved in a single search may be only 200 milliseconds, after you add the 1.7 seconds of startup processor time and then assume a 4:1 elapsed/processor time ratio, it will be 8 seconds before any response is printed.

3. Selecting and Formatting References for TROFF

The major application of the retrieval software is *refer,* which is a *troff* preprocessor like *eqn*. kernighan cherry acm 1975 It scans its input looking for items of the form

```
.[
imprecise citation
.]
```

where an imprecise citation is merely a string of words found in the relevant bibliographic citation. This is translated into a properly formatted reference. If the imprecise citation does not correctly identify a single paper (either selecting no papers or too many) a message is given. The data base of citations searched may be tailored to each system, and individual users may specify their own citation files. On our system, the default data base is accumulated from the publication lists of the members of our organization, plus about half a dozen personal bibliographies that were collected. The present total is about 4300 citations, but this increases steadily. Even now, the data base covers a large fraction of local citations.

For example, the reference for the *eqn* paper above was specified as

```
...
preprocessor like
.I eqn.
.[
kernighan cherry acm 1975
.]
It scans its input looking for items
...
```

This paper was itself printed using *refer.* The above input text was processed by *refer* as well as *tbl* and *troff* by the command

refer memo-file | tbl | troff –ms

and the reference was automatically translated into a correct citation to the ACM paper on mathematical typesetting.

The procedure to use to place a reference in a paper using *refer* is as follows. First, use the *lookbib* command to check that the paper is in the data base and to find out what keys are necessary to retrieve it. This is done by typing *lookbib* and then typing some potential queries until a suitable query is found. For example, had one started to find the *eqn* paper shown above by presenting the query

```
$ lookbib
kernighan cherry
(EOT)
```

lookbib would have found several items; experimentation would quickly have shown that the query given above is adequate. Overspecifying the query is of course harmless. A particularly careful reader may have noticed that ‘‘acm’’ does not appear in the printed citation; we have supplemented some of the data base items with common extra keywords, such as common abbreviations for journals or other sources, to aid in searching.

If the reference is in the data base, the query that retrieved it can be inserted in the text, between **.[** and **.]** brackets. If it is not in the data base, it can be typed into a private file of references, using the format discussed in the next section, and then the **−p** option used to search this private file. Such a command might read (if the private references are called *myfile*)

refer −p myfile document | tbl | eqn | troff −ms . . .

where *tbl* and/or *eqn* could be omitted if not needed. The use of the *−ms* macros lesk typing documents unix gcos or some other macro package, however, is essential. *Refer* only generates the data for the references; exact formatting is done by some macro package, and if none is supplied the references will not be printed.

By default, the references are numbered sequentially, and the *−ms* macros format references as footnotes at the bottom of the page. This memorandum is an example of that style. Other possibilities are discussed in section 5 below.

4. Reference Files.

A reference file is a set of bibliographic references usable with *refer.* It can be indexed using the software described in section 2 for fast searching. What *refer* does is to read the input document stream, looking for imprecise citation references. It then searches through reference files to find the full citations, and inserts them into the document. The format of the full citation is arranged to make it convenient for a macro package, such as the *−ms* macros, to format the reference for printing. Since the format of the final reference is determined by the desired style of output, which is determined by the macros used, *refer* avoids forcing any kind of reference appearance. All it does is define a set of string registers which contain the basic information about the reference; and provide a macro call which is expanded by the macro package to format the reference. It is the responsibility of the final macro package to see that the reference is actually printed; if no macros are used, and the output of *refer* fed untranslated to *troff,* nothing at all will be printed.

The strings defined by *refer* are taken directly from the files of references, which are in the following format. The references should be separated by blank lines. Each reference is a sequence of lines beginning with **%** and followed by a key-letter. The remainder of that line, and successive lines until the next line beginning with **%**, contain the information specified by the key-letter. In general, *refer* does not interpret the information, but merely presents it to the macro package for final formatting. A user with a separate macro package, for example, can add new key-letters or use the existing ones for other purposes without bothering *refer.*

The meaning of the key-letters given below, in particular, is that assigned by the *−ms* macros. Not all information, obviously, is used with each citation. For example, if a document is both an internal memorandum and a journal article, the macros ignore the memorandum version and cite only the journal article. Some kinds of information are not used at all in printing the reference; if a user does not like finding references by specifying title or author keywords, and prefers to add specific keywords to the citation, a field is available which is searched but not printed (**K**).

The key letters currently recognized by *refer* and *−ms,* with the kind of information implied, are:

center; c c6 c c c l c l. Key Information specified Key Information specified A Author's name N Issue number B Title of book containing item O Other information C City of publication P Page(s) of article D Date R Technical report reference E Editor of book containing item T Title G Government (NTIS) ordering number V Volume number I Issuer (publisher) J Journal name K Keys (for searching) X or L Label Y or M Memorandum label Z Information not used by *refer*

For example, a sample reference could be typed as:

```
%T Bounds on the Complexity of the Maximal
Common Subsequence Problem
%Z ctr127
%A A. V. Aho
%A D. S. Hirschberg
%A J. D. Ullman
%J J. ACM
%V 23
%N 1
%P 1-12
%M abcd-78
%D Jan. 1976
```

Order is irrelevant, except that authors are shown in the order given. The output of *refer* is a stream of string definitions, one for each of the fields of each reference, as shown below.

```
.]-
.ds [A authors' names ...
.ds [T title ...
.ds [J journal ...
...
.] [ type-number
```

The special macro **.]–** precedes the string definitions and the special macro **.] [** follows. These are changed from the input **.[** and **.]** so that running the same file through *refer* again is harmless. The **.]–** macro can be used by the macro package to initialize. The **.] [** macro, which should be used to print the reference, is given an argument *type-number* to indicate the kind of reference, as follows:

center; c c n l. Value Kind of reference 1 Journal article 2 Book 3 Article within book 4 Technical report 5 Bell Labs technical memorandum 0 Other

The reference is flagged in the text with the sequence

```
\* ([.number\* (.]
```

where *number* is the footnote number. The strings **[.** and **.]** should be used by the macro package to format the reference flag in the text. These strings can be replaced for a particular footnote, as described in section 5. The footnote number (or other signal) is available to the reference macro **.] [** as the string register **[F**.

In some cases users wish to suspend the searching, and merely use the reference macro formatting. That is, the user doesn't want to provide a search key between **.[** and **.]** brackets, but merely the reference lines for the appropriate document. Alternatively, the user can wish to add a few fields to those in the reference as in the standard file, or override some fields. Altering or replacing fields, or supplying whole references, is easily done by inserting lines beginning with **%**; any such line is taken as direct input to the reference processor rather than keys to be searched. Thus

```
.[
key1 key2 key3 ...
%Q New format item
%R Override report name
.]
```

makes the indicates changes to the result of searching for the keys. All of the search keys must be given before the first **%** line.

If no search keys are provided, an entire citation can be provided in-line in the text. For example, if the *eqn* paper citation were to be inserted in this way, rather than by searching for it in the data base, the input would read

```
...
preprocessor like
.I eqn.
.[
%A B. W. Kernighan
%A L. L. Cherry
%T A System for Typesetting Mathematics
%J Comm. ACM
%V 18
%N 3
%P 151-157
%D March 1975
.]
It scans its input looking for items
...
```

This would produce a citation of the same appearance as that resulting from the file search.

As shown, fields are normally turned into *troff* strings. Sometimes users would rather have them defined as macros, so that other *troff* commands can be placed into the data. When this is necessary, simply double the control character **%** in the data. Thus the input

```
.[
%V 23
%%M
Bell Laboratories,
Murray Hill, N.J. 07974
.]
```

is processed by *refer* into

```
.ds [V 23
.de [M
Bell Laboratories,
Murray Hill, N.J. 07974
..
```

The information after **%%M** is defined as a macro to be invoked by **.[M** while the information after **%V** is turned into a string to be invoked by ***([V**. At present *–ms* expects all information as strings.

5. Collecting References and other Refer Options

Normally, the combination of *refer* and *–ms* formats output as *troff* footnotes which are consecutively numbered and placed at the bottom of the page. However, options exist to place the references at the end; to arrange references alphabetically by senior author; and to indicate references by strings in the text of the form [Name1975a] rather than by number. Whenever references are not placed at the bottom of a page identical references are coalesced.

For example, the **–e** option to *refer* specifies that references are to be collected; in this case they are output whenever the sequence

```
.[
$LIST$
.]
```

is encountered. Thus, to place references at the end of a paper, the user would run *refer* with the *–e* option and place the above $LIST$ commands after the last line of the text. *Refer* will then move all the references to that point. To aid in formatting the collected references, *refer* writes the references preceded by the line

```
.]<
```

and followed by the line

```
.]>
```

to invoke special macros before and after the references.

Another possible option to *refer* is the **–s** option to specify sorting of references. The default, of course, is to list references in the order presented. The **–s** option implies the **–e** option, and thus requires a

```
.[
$LIST$
.]
```

entry to call out the reference list. The **–s** option may be followed by a string of letters, numbers, and '+' signs indicating how the references are to be sorted. The sort is done using the fields whose key-letters are in the string as sorting keys; the numbers indicate how many of the fields are to be considered, with '+' taken as a large number. Thus the default is **–sAD** meaning ''Sort on senior author, then date.'' To sort on all authors and then title, specify **–sA+T**. And to sort on two authors and then the journal, write **–sA2J**.

Other options to *refer* change the signal or label inserted in the text for each reference. Normally these are just sequential numbers, and their exact placement (within brackets, as superscripts, etc.) is determined by the macro package. The **–l** option replaces reference numbers by strings composed of the senior author's last name, the date, and a disambiguating letter. If a number follows the **l** as in **–l3** only that many letters of the last name are used in the label string. To abbreviate the date as well the form **-l**_m,n_ shortens the last name to the first _m_ letters and the date to the last _n_ digits. For example, the option **–l3,2** would refer to the _eqn_ paper (reference 3) by the signal _Ker75a_, since it is the first cited reference by Kernighan in 1975.

A user wishing to specify particular labels for a private bibliography may use the **–k** option. Specifying **–k**_x_ causes the field _x_ to be used as a label. The default is **L**. If this field ends in –, that character is replaced by a sequence letter; otherwise the field is used exactly as given.

If none of the _refer_-produced signals are desired, the **–b** option entirely suppresses automatic text signals.

If the user wishes to override the _–ms_ treatment of the reference signal (which is normally to enclose the number in brackets in _nroff_ and make it a superscript in _troff_) this can be done easily. If the lines **.[** or **.]** contain anything following these characters, the remainders of these lines are used to surround the reference signal, instead of the default. Thus, for example, to say ''See reference (2).'' and avoid ''See reference.[2]'' the input might appear

```
See reference
.[ (
imprecise citation ...
.]).
```

Note that blanks are significant in this construction. If a permanent change is desired in the style of reference signals, however, it is probably easier to redefine the strings **[.** and **.]** (which are used to bracket each signal) than to change each citation.

Although normally _refer_ limits itself to retrieving the data for the reference, and leaves to a macro package the job of arranging that data as required by the local format, there are two special options for rearrangements that can not be done by macro packages. The **–c** option puts fields into all upper case (CAPS-SMALL CAPS in _troff_ output). The key-letters indicated what information is to be translated to upper case follow the **c**, so that **–cAJ** means that authors' names and journals are to be in caps. The **–a** option writes the names of authors last name first, that is _A. D. Hall, Jr._ is written as _Hall, A. D. Jr_. The citation form of the *Journal of the ACM*, for example, would require both **−cA** and **−a** options. This produces authors' names in the style *KERNIGHAN, B. W. AND CHERRY, L. L.* for the previous example. The **−a** option may be followed by a number to indicate how many author names should be reversed; **−a1** (without any **−c** option) would produce *Kernighan, B. W. and L. L. Cherry*, for example.

Finally, there is also the previously-mentioned **−p** option to let the user specify a private file of references to be searched before the public files. Note that *refer* does not insist on a previously made index for these files. If a file is named which contains reference data but is not indexed, it will be searched (more slowly) by *refer* using *fgrep.* In this way it is easy for users to keep small files of new references, which can later be added to the public data bases.