Combining the Bourne-shell, sed and awk in the UNIX environment for language analysis

Lothar M. Schmitt[1] and Kiel T. Christianson[2]

[1]The University of Aizu, School of Computer Science and Engineering,
Aizu-Wakamatsu City, Fukushima Prefecture, 965-80, Japan. E-mail: lothar@u-aizu.ac.jp
[2]Michigan State University, Dept. of Linguistics and Germanic, Slavic, Asian and African Languages, East Lansing, Michigan 48824, USA. E-mail: chris118@pilot.msu.edu
[1,2] Both authors are equally main contributors to this publication. The authors are listed in non-alphabetical order due to evaluation procedures within The University of Aizu which do not recognize a main contributor properly unless listed as first author.

Abstract

We show how to construct tools for language analysis in research and teaching using the Bourne-shell, sed and awk under UNIX. Applications include the following: searches for words, phrases, grammatical patterns and phonemic patterns in text; statistical evaluation of texts in regard to such searches; transformation of phonetic, phonemic or typographic transcriptions; comparison of texts in various respects; lexical-etymological analysis; concordance; assistance in translating text; assistance in learning languages; assistance in teaching languages; and text processing and formatting. The latter includes the generation of on-line dictionaries for the internet from files that were generated with what-you-see-is-what-you-get editors representing only the linear structure of the dictionary (i.e., the book). All of the above can be achieved with particularly simple and short code. In that regard, we illustrate how sed and awk can be combined in the pipe mechanism of UNIX to create very powerful processing devices. Our notes include a short introduction to programming the Bourne-shell and rather short, but complete descriptions of sed and awk customized in regard to language analysis.

1 Introduction

This paper presents an outline of the rich variety of applications to language analysis that opens up through the combined use of two simple yet powerful programming languages with particularly short descriptions: `sed` and `awk`.

`sed` and `awk` are standard tools under UNIX. Some introductions to their use can be found in rather complete introductions to UNIX, e.g., [Kernighan & Pike 1984]. This may create the false impression that a broad understanding of UNIX is necessary in order to use `sed` and `awk` effectively for language analysis and other purposes. Some introductions are devoted to just one of the two, e.g., [Aho et al. 1978]. This does not explore the possibilities that open up through the combined use of `sed` and `awk` in the pipe mechanism of UNIX. However, [Aho et al. 1978] is a very good, compact reference. Some other introductions are rather long, e.g., [Dougherty 1990]. This may create the false impression that it takes much time and effort to learn `sed` and `awk`.

In order to use `sed` and `awk` effectively for language analysis, only a small amount of UNIX is needed. In the first part of these notes, we give a very short, complete and convenient introduction to the use of `sed` and `awk` for language research and related applications. We have listed only the minimal number of facts needed to write programs for the Bourne-shell `sh` such that `sed` and `awk` can be combined in the pipe mechanism of UNIX to create very powerful processing devices. The examples given are procedures which are particularly useful for language processing. They are listed ready to use. Introducing `sed` and `awk` at the same time makes it possible to present pattern matching, their main common feature, quite effectively. It is only assumed that the reader is familiar with a text editor on a workstation such as `emacs`, `mule`, or `vi`. `emacs` and `mule` are easy to use through their pull down menus, which are self-explanatory and the built-in tutorial.

Essentially, `sed` and `awk` operate on lines of input. A line of input coming from a text file or the UNIX pipe mechanism is manipulated and, usually, delivered further into the pipe or to a final output file. It is a simple, convenient and powerful philosophy to let the flow of data in the pipe be the storage facility for entities in text that have been recognized, put on different lines and, possibly, marked by appropriately chosen keywords. To illustrate this idea suppose that text in a file can be rearranged by one program in such a way that every line of output contains exactly one sentence. Such a process of reformatting text can easily be used in a pipe to achieve many objectives. These include counting words in sentences (an additional one-line `awk` program counting the fields, i.e., strings of non-white characters containing letters in a line) or searching for sentences that contain a specific word (an additional one-line `sed` or `awk` program). Principle tasks for `sed` are substitution using the tagged regular expression mechanism and pattern matching for the purpose of isolating certain lines. Principle tasks for `awk` are pattern matching for the purpose of isolating certain lines, pattern matching that needs numerical computation, accounting and rearranging fields within a line. However, it is really the combination of `sed` and `awk` that makes many very

simple solutions for complicated tasks possible. For example, it is easy to remove punctuation marks from text and to map to lower case using `sed`. On the other hand, counting is simple in `awk`. Consequently, combining the two makes it easy to implement a word frequency count. `sed` and `awk` programs should be rather short and well commented. Complicated tasks should be solved though a combination of filters in a pipe each doing a simple job.

Included in our detailed description of `sed` and `awk` is a collection of ideas and methods that should enable people to write short, customized applications for language analysis combining the simplicity and the potential of the two programming languages. We shall show that by combining basic components each containing a few lines of code one can generate a flexible and powerful customized environment. In addition, more elaborate tools such as `lex` [Lesk & Schmidt 1978] [Kernighan & Pike 1984] or `yacc` [Johnson 1978] [Kernighan & Pike 1984], and programming languages such as `C` [Kernighan & Ritchie 1988] or `prolog` [Clocksin & Mellish 1981] can be used together with the methods presented here. In particular, note that, e.g., `mathematica` [Wolfram 1991] can be used to generate graphics from numerical data produced with `awk`.

The later sections of these notes describe methods of application. Some of these application have been used in [Schmitt & Christianson 1998] which is a system designed to support the teaching of English as a second language by computer means. In particular, we have used `sed` and `awk` for analysis of short essays that were submitted as homework by Japanese students of English composition via e-mail. Implementations that will be described in these notes are a punctuation checker and a program that reformats text in such a way that whole sentences are on single lines. A punctuation checker is obviously well-suited for automated evaluation, correction and return. The second program is, as already indicated above, well-suited for all sorts of subsequent selection schemes which, usually, can be implemented with simple additional means. It can be used to select phrases and sentences that were submitted by students and contain critical or interesting grammatical patterns for presentation in class. Another example of application is the analysis of grammatical patterns in students' writings and their statistical evaluation. We use `sed` in [Schmitt & Christianson 1998] to implement various selection and tagging schemes. `awk` is used in [Schmitt & Christianson 1998] to implement set and vector operations (which can be used, e.g., to measure the increase of vocabulary used by students) and elementary statistical operations. It was important during the project [Schmitt & Christianson 1998] to have very flexible tools such as `sed` and `awk` available through which a collection of experimental programs could be implemented fast and altered easily. Components that have proven to be useful are later rewritten in `C` in order to shorten execution time.

In addition to the applications just described, we show how to set up a vocabulary training environment, how to develop tools for statistical evaluation of text, be it in regard to concordance (cf. [Kennedy 1991] and [Renouf & Sinclair 1991]), in regard to lexical-etymological analysis (cf. [Gordon 1996]), or in regard to judging the readability of text (cf. [Hoey 1991]).

In [Kennedy 1991], an analysis of collocations occurring with "between" and "through" was conducted with "The Oxford Concordance Program OCP2" [Hockey & Martin 1988]. We shall explicitly show how a search for such collocations can be implemented using `sed` and `awk` with a few lines of easy-to-understand and easy-to-costumize code. In [Renouf & Sinclair 1991], a corpus search for the strings `a...of`, `an...of`, `be...to`, `too...to`, `for...of`, `had...of` and `many...of` was conducted. Such a search including the sorting of the results into separate files can also be implemented with a few lines of code. We shall describe how to implement a lexical-etymological analysis on a machine as done in [Gordon 1996] by hand. And, we shall describe how our procedure which counts word frequencies can be used to roughly judge the readability of text (cf. [Hoey 1991]). Finally, we shall indicate how `sed` and `awk` can be used to implement special parsers that transform a linear source file for a dictionary (here: [Nelson 1962]) into a multi-dimensional database for the internet. In addition, our exposition contains many comments in regard to other application using particular features of `sed` and `awk` such as identifying Japanese kanji characters in bilingual text or assisting translation.

As outlined above, we present a particularly short and customized introduction to the use of `sed` and `awk` under UNIX in language research including a large variety of applications. Such an approach is rarely found in the literature even though the two programming languages are classical tools and have been documented in detail for other audiences (cf. [Kernighan & Pike 1984], [Dougherty 1990], [Herold 1994]). Scattered reference to `sed` and `awk` can be found in descriptions of literary computing, e.g., [Goldfield 1986], who uses the tools for literary computing in French. However, we are not aware of any presentation of `sed` and `awk` geared toward linguistic analysis that is as short, detailed and complete as the following. We shall demonstrate that `sed` and `awk` provide easy-to-understand means to use programming in linguistic research. An alternative such as `prolog` forces a programming style which some may find counter-intuitive. In addition, `prolog` needs a "clean" formatted source, i.e., a `prolog` database. Another alternative, using `lex` and `yacc` requires formulating actions with `C` routines. Thus, a detailed understanding of the latter machine-close programming language and compiling are necessary. However, `prolog` and `yacc` are clearly better suited for elaborate grammatical analysis than `sed` and `awk`. A genuine alternative to the approach presented in this paper is using `perl` [Wall & Schwarz 1990].

Some people criticize that there is some overlap in the capabilities of `sed` and `awk` while on the other hand there are discrepancies in the notation for both. For example, there is no repitor `+` for regular expressions in `sed`. Observe that `sed` and `awk` were invented at different times by different people with different objectives and personal tastes. In fact, `sed` is older than `awk`. One should accept both tools as being simple, easy to learn, and extremely useful, and one should recognize the amount of work that went into creating both rather than critize small discrepancies.

Finally, note that the tools `sh`, `sed` and `awk` which we have used here as well as the

pipe mechanism are also available for other operating systems. Consequently, the methods presented here can easily be ported to platforms where these means are available.

2 A word frequency count

Let us show that "combining `sed` and `awk` makes it easy to implement a word frequency count." Essentially, this is done by the following complete program for the Bourne shell:

```
sed  'y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/
s/[^A-Za-z][^A-Za-z]*/\
/g'  fName     |     awk  '{ n[$0]++ }
                     END   { for (word in n) { print word , n[word] }  }'
```

The `sed` command `y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/` maps upper case letters to lower case. The `sed` command `s/[^A-Za-z][^A-Za-z]*/\ /g` substitutes every string of non-letters by a newline character. Consequently, it puts all strings of letters on separate lines. This procedure is applied to the file `fName`. The output of the `sed` program containing single words on lines is then fed as input into the `awk` program using the pipe symbol `|`. The `awk` statement `{ n[$0]++ }` increments a counter variable `n[`*word*`]` by 1, if *word*=`$0` is found on the line. Every occurring `n[`*word*`]` is automatically initiated to 0. The `awk` statement in the last line prints the list of type/token ratios for the words which were encountered. All the above will be explained below in greater detail including some improvements such as sorting the result and handling of words that contain a hyphen.

3 Programming the Bourne-shell

The next section contains a minimal collection of facts needed to use the Bourne-shell `sh` on a UNIX workstation. Readers familiar with `sh` may wish to skip it.

3.1 The manual pages

The most useful command in UNIX is the manual pages command `man`. If you are sitting in front of a UNIX terminal, type `man csh`↩ (where ↩ represents a carriage return). This will copy a detailed description of the c-shell `csh` to the terminal. You are most likely using `csh` right now in your terminal. Hitting the space-bar on the keyboard advances through the description. Typing Control-`c` (interrupt, here: interrupt execution of the `man` command)

terminates `man csh` before the end of the description is reached. You do not need to work through this description in order to understand the remainder of this paper. `csh` is very well suited for interacting with the UNIX environment via the terminal. In what follows, we shall usually not mention typing carriage returns. Consult `man man` for further details about `man`. (*Note:* The manual pages are divided into several volumes. Usually, `man` accesses volume 1. For example, `man 3 printf` accesses the content about `printf` in volume 3.)

3.2 Creating a UNIX command using the Bourne-shell

One can activate the Bourne-shell `sh` by typing `sh`↩ in a terminal which uses `csh`. `sh` works with the same UNIX as `csh` but there are some minor notational differences for the use of UNIX within this shell. `sh` presents itself with a `$` as prompt. It is very important to note that in this state one can test programs for `sh` interactively line-by-line in the terminal. Typing Control-`d` (end-of-file, here: "end of the list of commands" that come from the terminal) in `sh` causes `sh` to terminate. For `sh`, a list of UNIX commands that it reads from a file or from a terminal window are indistinguishable. In the remainder of this section, we shall discuss how to set up programs for `sh`. Essentially, a `sh` program is a file containing one `sh` command per line as well as multi-line commands. These commands are worked through by `sh` from top to bottom in the file. Several (single-line) commands can also be separated by semicolons `;` and listed on one line. In that case, they are executed from left to right in the line.

Example: Copy the following four lines into a file `aizu` with your favorite text editor:

```
#!/bin/sh
# Comment:  Aizu Telephone Numbers
echo  'aizu phone  +81-242-37-2500'
echo  'aizu fax    +81-242-37-2528'
```

Save this file as `yourLoginName/aizu` directly in your home directory and not a subdirectory. After having done that, type `cd ; chmod 700 aizu` in your currently used shell. `cd` sets your working directory to your home directory. `chmod 700 aizu` makes `aizu` executable for you in addition of being readable and writable. Consult `man cd` and `man chmod` for further details about `cd` and `chmod`. `aizu` is now a UNIX command just like the built-in ones.

♡ Note that the few simple steps just described are all you need to know in order to make an executable file of `sh` commands!

Next, type `aizu` in your shell to see what the program does. The first line `#!/bin/sh` of `aizu` tells whatever shell you are using that the command lines are designed for `sh` which executes the file. Thus, you do not need to switch to `sh` in order to execute a command file for `sh`. The second line is comment. Comment for `sh`, `sed` and `awk` starts by definition with a `#` as first character in a line. Note, that comment usually does not work within, e.g., a

multi-line `sed` command. The `echo` command does what its name says: it prints the strings of characters that follow on the same line (until the end of the line or a semicolon) plus an appended newline character. Actually, an on-line notebook as `aizu` at the workplace is quite convenient. Compare the discussion of `cat` given below.

3.3 Strings

Strings are framed by single quotes `'` and are separated by white space consisting of blanks or tabs. Strings can be concatenated: `echo 'a$''>0'` produces `a$>0`. Usually, the framing pair of single quotes `'` can be omitted, if the string consists of letters and digits only. If the single quote `'` is supposed to be included as character in a string, then it has to be represented as `\'` outside of other framing single quotes. To see that this works, try: `echo ''‘''ECu\'`.

3.4 Pipes

Previously defined UNIX commands can be used in the design of new commands. The following example shows how to connect the output of `aizu` with another command (`grep`) in the pipe mechanism of UNIX:

```
#!/bin/sh
aizu  |  grep fax -
```

Save these two lines as `aizufax` in your home directory, make the file executable (using `cd; chmod 700 aizufax`), and see what it does! `grep` is a simple filter that isolates the lines in its input that contain the given pattern (here: `fax`). The important point in the above `sh` program is the single vertical slash `|`. `|` tells `sh` to use the output of `aizu` as input for `grep`. The combination of `aizu` with `grep fax -` in this way is called a pipe which is seen by `sh` as a single UNIX command. One can connect any number of commands through `|` in a pipe. The hyphen `-` used above stands for the (virtual) input file of `grep` in the pipe. In UNIX terminology, it is called *stdin* (standard input). In many cases, it can be omitted. If you are in doubt, then just include it. Usually, `sh` interprets reasonable expressions in a reasonable way.

For better readability of programs, one may wish to spread pipes over several lines. The following example does exactly the same as `aizufax`:

```
#!/bin/sh
aizu |\
grep x
```

The backslash[1] character \ with nothing (not even a blank or tab) following tells `sh` that the command is continued in the next line. The pattern `x` is sufficient to identify the second line of output of `aizu`, and the hyphen denoting *stdin* is omitted.

One application of pipes is to use `sed` and `awk` in preprocessing a text file which is then piped into a text formatting program such as the UNIX built-in `nroff` [Ossanna & Kernighan 1978] or `latex` [Lamport 1986]. `nroff` is particularly simple to use and sufficient for many purposes.

3.5 Arguments to a command

The following example shows how to use three arguments `$1`, `$2` and `$3` for a command:

```
#!/bin/sh
echo  $3 $1 $2
```

Save the above as `cab` and make it executable (using `chmod 700 cab`). Invoke this command with
`cab  Apes Bees  Cacadus  Dinos` and with
`cab 'Apes Bees  Cacadus' Dinos`.
In `cab  Apes Bees  Cacadus  Dinos`, the first argument `$1` is the string `Apes`, the second argument `$2` is the string `Bees`, and the third argument `$3` is the string `Cacadus`. `echo` echoes accordingly. In `cab 'Apes Bees  Cacadus' Dinos`, the first argument `$1` is the string `Apes Bees  Cacadus`, the second argument `$2` is the string `Dinos`, and there is no third argument `$3`.

Usually, a UNIX command does something with a file. In that case, the name of the file is one of the arguments that is passed on. Up to nine arguments `$1`...`$9` can be used. Arguments to a UNIX command are strings separated by white space consisting of blanks or tabs.

3.6 Diverting output to a file

Usually, the output of a UNIX command is channeled into a file and not to the terminal. The following example shows how this is achieved. Type the following four command lines in a shell (without saving them to a file):

```
sh
echo  'aizu   http://www.u-aizu.ac.jp/'  >www
echo  'yahoo  http://www.yahoo.com/'     >>www
```

[1] Actually, the backslash "escapes" the immediately following newline character, i.e., the latter is understood by `sh` as a character and not as a command-line terminator. For `sh`, the function of the (literal) newline character is that of the blank if it is not embedded in a string.

```
cat www
```

`>www` diverts the output of `echo` to the file `www` which is created (or possibly overwritten). `>>www` appends the output of `echo` to the file `www`. If a file who's name is given does not exist, then `>>` creates that file too. `cat www` copies the file `www` to the terminal. Moreover, `cat file1 file2 ...` concatenates given files. The result can, e.g., be delivered into a UNIX pipe. Consult `man cat` for more details.
(*Note:* The UNIX command `echo >fname` creates an empty file named `fname`.)

3.7 Never divert output to your source file!

More information about programming `sh` and the UNIX commands mentioned above can be obtained using the `man` command as well as consulting [Kernighan & Pike 1984].

4 sed

4.1 Global substitution

`sed` is the ideal tool to make replacements in texts. A `sed` program operates on a file line-by-line. If nothing is done with a line, then it is simply copied. The following example shows a `sh` program that replaces the patterns `george` and `NEWLINE` in all instances in a file `$1` by `bill` and a newline character, respectively:

```
#!/bin/sh
sed  's/george/bill/g
s/NEWLINE/\
/g'  $1
```

The two single quotes ' tell `sh` where the string that constitutes the `sed` program starts and ends. These are delimiters for `sh` and are not communicated to `sed`.

The `sed` command `s/george/bill/g` consists of four parts:
1) `s` is the `sed` command used and stands for "substitute."
2) `george` is the pattern that is to be substituted.
3) `bill` is the replacement for the pattern.
4) The `g` means "global." Without the `g` only the first occurrence of the pattern would be replaced in a line.

The second substitution command shows the important technique of how to place newline characters at specific places. There is nothing following the backslash \ which is part of the `sed` program.

The argument `$1` is supposed to be the name of the text file the `sed` program is applied to.

In order to use the above `sh` program, save it as `pce` and make it executable. Then, do the following in a shell:

```
echo  george noNEWLINE georges  >9294  ;    pce 9294
```

Application: With a one-line `sed` substitution as above, all occurrences of something that should be maintained only at one place (e.g, `OUR_ADDRESS`) can automatically be (re)placed in the source file of a document which is preprocessed with `sed`.

Application: With a `sed` program similar to the above, one can reformat text which uses a non-standard phonetic, phonemic, or typographic transcription. In such a program, one is not limited to global substitution commands: exceptional cases can also be incorporated into the program using appropriately chosen pattern matching or addresses (see below). Such a reformatted text can then be compared to other sources in regard to use of words, phrases and morpho-phonemic structure. At least in part, this can be also done by machine using `sed` and `awk`. The latter sort of analysis will be outlined below in more detail. With a `sed` program similar to the above, it is also possible to convert phonetic transcription alphabets, e.g., 'IPA' to 'SIL.'

4.2 Some additional comments

`sed` commands are terminated by either an immediately following newline character or a semicolon or the end of the program. If a `sed` command is terminated by a semicolon, then the next `sed` command can follow on the same line. There is one exception to this: the `w` command (write) which is explained below. After a `w` command and some separating white space, everything that follows on the same line is understood as the filename to which the command is supposed to write.

One can also store the `sed` commands listed above without the two single quotes but with the backslash in a file (say) `sedCommands` and use `sed -f sedCommands` instead of or in the above `sh` program. Observe that if a `sed` program is used with a separate file of commands in a UNIX pipe, then this makes reading an additional file necessary. This may slow down the overall process.

One may want to process the single quote `'` in `sed` programs. The following `sh` program shows how to replace all single quotes `'` in a file `$1` by the string `QQ`. `'\''` can always be used to include the single quote in a program.

```
#!/bin/sh
sed  's/'\''/QQ/g'  $1
```

The program uses two strings `'s/'` and `'/QQ/g'` and concatenates those with `\'` in which the single quote is delivered to `sh` as itself using the escape character backslash `\`. The combined string `s/'/QQ/g` is then communicated by `sh` to `sed`. `s/'/QQ/g` is what one would

put in a separate file of `sed` commands. The representation of the slash `/` and backslash characters in `sed` programs is explained below.

The replacement in a substitution can be empty. This can, e.g., be used to "clean" a `tex` file from control sequences.

4.3 The pattern space and the hold space

`sed` keeps track of two "buffers." The first one is the pattern space. Roughly speaking, it can be seen as the current, possibly already altered line. More precisely, every line of input is put into the pattern space and is worked on therein by every command line of the entire `sed` program from top to bottom. This is called a *cycle*. After the cycle is over, the resulting pattern space is printed. Lines that were never worked on are consequently copied by `sed`. Each `sed` command that is applied to the content of the pattern space may alter it. In that case, the previous version of the content of the pattern space is lost. Subsequent `sed` commands are always applied to the current content of the pattern space and not the original input line. Using `sed` with the option "`-n`" (i.e., using `sed -n`) switches the final printing off. There is a separate print command `p` for printing the pattern space.

The second buffer used by `sed` is the hold space. The pattern space can be stored in the hold space for, e.g., printing which depends upon further processing or repeated analysis. The hold space is not erased if a new cycle is begun. The content of the pattern space can be overwritten by the content of the hold space. In addition, appending one of the two buffers to the other is possible.

If `pce` is applied to a file, then, first, all strings `george` in a line are replaced by strings `bill`. Second, while the possibly altered line is still in the pattern space, all strings `NEWLINE` are replaced by newline characters. These two actions comprise the cycle per line.

4.4 Format of `sed` commands

The format of a `sed` command is:

Address`Command`

Address can be omitted. In that case, `Command` is then applied to every pattern space. If an *Address* is given, then `Command` is applied to the pattern space only in the case the latter matches *Address*.

Example: The following program replaces `TX` with `Texas` in all lines that contain the string `USA`.

```
#!/bin/sh
sed  '/USA/s/TX/Texas/g'  $1
```

4.5 Patterns (regular expressions)

Patterns which are also called regular expressions can be used in `sed` for two purposes:
1) As addresses, in order to select the pattern space for processing by `sed` commands.
2) As patterns in substitution commands that are actually replaced.

Patterns are matched by `sed` as the longest, non-overlapping strings possible. The patterns that can be used consist of the following elements in between slashes `/`:
1) Any non-special character matches itself.
Examples: `/thing/` resp. `/aZ9/` match the string `thing` resp. `aZ9`. `/thing/` as an address would cause the pattern space to be selected, if the latter contained, e.g., `anything`. If `/thing/` were the pattern in a substitution command, then it would cause a substitution within, e.g., `nothing`.
2) Special characters that otherwise have a particular function in `sed` have to be preceeded by a backslash `\` in order to be understood literally. As already illustrated above, some of the `sed` commands allow the placement of newline characters in the pattern space. Such newline characters can be matched with `\n`. However, `\n` does not match the beginning or end of the pattern space.
All special characters: `\\ \/ \^ \$ \. \[ \] \* \& \n`
(*Note:* In the replacement in a substitution command, only `\`, `/` and `&` have to be preceeded by a backslash in order to be understood literally, and `\n` evaluates to `n`.)
Example: `sed 's/CAN\$/US$/g'` changes `CAN$` to `US$`.
3) `^` resp. `$` match the beginning resp. the end of the pattern space. `^` and `$` should be seen as markers of length 0 rather than as characters of length 1. They must not be repeated in the replacement in a substitution command.
Examples: `sed 's/^Chicken/Hawks/'` replaces `Chicken` at the beginning of lines with `Hawks`. `sed 's/$/0/'` appends a zero to the right of every line in a file. `/^$/` matches the empty pattern space.
4) `.` matches any single character.
Example: `sed 's/..//'` removes the first two characters in every line of a file containing more than one character. `/./` as an address selects the non-empty pattern space.
5) `[`*what*`]` matches any character in *what* where *what* is a string of characters. The following five rules must be observed:
R1: The backslash `\` is not needed and used to indicate special characters in *what*. The backslash only represents itself.
R2: The closing bracket `]` must be the first character in *what* in order to be recognized as itself.
R3: Ranges of the type `a-z`, `A-Z`, `0-9` in *what* are permitted.
R4: The hyphen `-` must be at the beginning or the end of *what* in order to be recognized as itself.
R5: The carat `^` must not be the first character in *what* in order to be recognized as itself.

Examples: `/[Tt]urkey/` matches `Turkey` and `turkey`. `/[]^K-M-]/` as an address selects every pattern space which contains `]`, `^`, `K`, `L`, `M` or `-`.
6) `[^`*what*`]` matches any character not in *what*. The rules **R1**–**R4** set under 5) also apply here.
Example: `/[^]^\2-5-]/` as an address selects every pattern space which contains something different from `]`, `^`, `\`, `2`, `3`, `4`, `5` and `-`.
7) *pattern*`*` stands for 0 or any number of copies of *pattern* where *pattern* is a specific character, the period `.` (meaning any character) or a range `[...]` as described under 5) or 6).
Examples: `/[a-zA-Z][a-zA-Z]*/` matches non-empty strings of letters having maximal possible length. `/..*[A-Z]/` matches the longest string possible that contains at least two characters and ends with a capital letter. Thus, as an address it matches every pattern space which contains a capital letter not as first character as does `/.[A-Z]/`.

As indicated in the last two examples, patterns are matched by `sed` as the longest, non-overlapping strings possible. If one wants to process overlapping pattern, then one can use the `t` command described below.

In the next sections, we shall explore the possibilities in using patterns in substitution commands. This is in our experience the most frequent use of patterns. Patterns as addresses and other types of addresses will be discussed afterwards.

4.6 Some simple preprocessing devices

The next simple examples show how text can be preprocessed with small, customized `sed` programs such that the output can conveniently be used for further processing in a pipe. Alternatively, the code given below may be included in larger `sed` programs when needed. However, dividing processes into small entities as given in the examples below is a very useful technique to isolate reusable components and to avoid programming mistakes resulting from over-complexity of single programs.

Example: The following `sh` program adjusts blanks and tabs in a file `$1` in such a way that it is better suited for certain searches. In what follows, we shall refer to this program as `addBlanks`. All ranges in the `sed` program contain a blank and a tab.

```
#!/bin/sh
sed  's/[  ][  ]*/  /g;    s/^[  ]*/ /;    s/[  ]*$/ /;    s/^[  ]*$//'  $1
```

First, all strings consisting only of blanks and tabs are replaced by two blanks. Then, a single blank is placed at the beginning and the end of the pattern space. Finally, any resulting white pattern space is cleared from blanks and tabs in the last substitution command.
Application: Suppose that one wants to search in a file for use of the word "liberal." In order to identify the strings `Liberal` and `liberal` in raw text properly, one needs the following four patterns:

```
/[^A-Za-z][Ll]iberal[^A-Za-z]/ /^[Ll]iberal[^A-Za-z]/
/[^A-Za-z][Ll]iberal$/         /^[Ll]iberal$/
```

The string `liberal liberal` which legally may occur in text shows that the first pattern must even be repeated, if, e.g., one would want to eliminate `liberal` from the file using `sed`. To identify the first `liberal`, `sed` needs the blank in the string which is then not available to identify the second. Recall that `sed` matches non-overlapping patterns. Instead of repeating the first pattern, one could loop over it once. (Looping with `sed` will be explained below.) If one preprocesses the source file with `addBlanks`, only the first pattern is needed once. Thus, a `sed` based search program for `Liberal` and `liberal` is shortened and faster.

Example: The following program is a variation of `addBlanks`. It can be used to isolate words in text in a somewhat crude fashion. In fact, abbreviations and words that contain a hyphen or an apostrophe are not properly identified. The white ranges in the `sed` program contain a blank and a tab each.

```
#!/bin/sh
sed  's/[^A-Za-z][^A-Za-z]*/  /g;     s/^[  ]*/ /
      s/[  ]*$/ /;                    s/^[  ]*$//'  $1
```

First, all strings consisting only of non-letters are replaced by two blanks. Then, a single blank is placed at the beginning and the end of a line. Finally, any resulting white pattern space is cleared from blanks and tabs in the last substitution command.

Example: The following `sh` program which removes obsolete blanks and tabs in a file `$1` is somewhat the inverse of `addBlanks`. In what follows, we shall refer to this program as `adjustBlankTabs`. Every range contains a blank and a tab.

```
#!/bin/sh
sed  's/^[  ]*//;    s/[  ]*$//;    s/[  ][  ]*/ /g'  $1
```

All leading and trailing white space (blanks and tabs) is removed by the first two substitution commands of the `sed` program. All white strings are replaced by a single blank in the last substitution command.

Application: `adjustBlankTabs` standardizes and minimizes phrases (as strings) which may automatically be obtained from e-mail messages with inconsistent typing style or text files that have been justified left and right. This is useful if one wants to analyze sentences and, e.g., derive statistics over phrases which are processed as unique strings of characters.

4.7 Using what was matched

The character `&` can be used in the replacement in a substitution command to reproduce the string that was matched in the pattern in the substitution command. `&` can be used repeatedly in order to reproduce the string that was matched.

Example: The following program folds all lines in a text (inserts newline characters) after the first string of blank or tabs following every string of at least 10 characters. All ranges contain a blank and a tab.

```
#!/bin/sh
sed  's/..........[^  ]*[  ][  ]*/&\
/g'  $1
```

`..........[^  ]*` in the pattern represents a string of at least 10 characters which is non-white after the 10^{th} character. `[  ][  ]*` in the pattern represents a subsequent string of blank or tabs. A newline character is inserted in the pattern space after every sequence of characters specified in the combined pattern.
Application: Some editors allow sending files via e-mail from within the editor. At the same time, they automatically fold lines on the sender's screen. This leads to particularly long lines in e-mail messages which, e.g., cannot be printed or may contain "too many fields" for `awk`. If one intends to process such e-mail messages automatically, then a customized version of the above program that folds after 120 characters can be used to counter this effect.

4.8 Tagged regular expressions

The tagged regular expression mechanism is the most powerful programming device in `sed`. It can be used for extending, deviding and rearranging patterns and their parts. Up to 9 chunks of the pattern in a substitution command can be framed (tagged) with `\(` and `\)`. For example, tagging the integer part in the pattern `/[0-9][0-9]*\.[0-9]*/` which matches numbers such as `90.9` or `4.` yields `/\([0-9][0-9]*\)\.[0-9]*/`. The recognized strings can be reused in the pattern *and* the replacement in the substitution command as `\1`, `\2`, `\3` ... counting from left to right. More detail about the usage of tagged regular expressions is given in the following examples:

Example: The following program shows a first application of the techniques introduced so far. It marks all determiners in a text file `$1`. We shall refer to it as `markDeterminers`.

```
#!/bin/sh
addBlanks $1 |\
sed  's/\.\.\./_TRIPLE_PERIOD_/g
s/\([[{(< `"_]\)\([Aa]\)\([]})> '\''",?!_]\)/\1_DETERMINER_\2_\3/g
s/\([[{(< `"_]\)\([Aa]\)\(\.[^A-Za-z]\)/\1_DETERMINER_\2_\3/g
s/\([[{(< `"_]\)\([Aa]n\)\([]})> '\''",?!_.]\)/\1_DETERMINER_\2_\3/g
s/\([[{(< `"_]\)\([Tt]he\)\([]})> '\''",?!_.]\)/\1_DETERMINER_\2_\3/g
s/\([[{(< `"_]\)\([Tt]hat\)\([]})> '\''",?!_.]\)/\1_DETERMINER_\2_\3/g
s/\([[{(< `"_]\)\([Tt]his\)\([]})> '\''",?!_.]\)/\1_DETERMINER_\2_\3/g
s/\([[{(< `"_]\)\([Tt]h[eo]se\)\([]})> '\''",?!_.]\)/\1_DETERMINER_\2_\3/g
s/_TRIPLE_PERIOD_/.../g'  |  adjustBlankTabs -
```

Explanation of the central `sed` program: The first substitution command replaces the triple period as, e.g., in "Bill bought...a boat and a car." by the marker `_TRIPLE_PERIOD_`. This distinguishes the period in front of "a" in the example given above from an abbreviation

such as "a.s.a.p." The character preceeding a determiner is encoded left in every pattern as `[[{(< ‘"_]`, tagged and reused as `\1` in the replacement in the substitution command. The determiner which is specified in the middle of every pattern is reused as `\2`. The non-letter following a determiner is encoded right in the last five patterns (for "An" through "those") as `[]})> ’\’’",?!_.]`, tagged and reused as `\3`. The string `’\’’` represents a single `’`. For the determiner "a" the period is excluded in the letters that follow it in `[]})> ’\’’",?!_]`. If a period follows "a," then a non-letter must follow. This is encoded in `\.[^A-Za-z]` in the fifth line of the program. Also the string encoded as `\.[^A-Za-z]` is tagged and reused as `\3`. After the tagging is completed, the triple period is restored. For example, the string `"A  liberal?"` is replaced by the program with `"_DETERMINER_A_  liberal?"`.
A note on `addBlanks`: Instead of using `addBlanks` one may be tempted to work, e.g., with the following substitution command

```
s/\([[{(< ‘"_]*\)\([Aa]n\)\([]})> ’\’’",?!_.]*\)/\1_DETERMINER_\2_\3/g
```

in order to deal with the cases when a determiner occurs at the beginning or end of a line. However, this substitution command causes the string `"Another?"` to be replaced by `"_DETERMINER_An_other?"`, i.e., the "empty" non-character string matched by `\([]})> ’\’’",?!_.]*\)` is properly processed in accordance with the given substitution command.
Application: A collection of tagging programs such as `markDeterminers` can be used for elementary grammatical analysis. If a file contains only whole sentences per line, then a pattern `/_DETERMINER_.*_DETERMINER_/` finds all sentences that contain at least two determiners. To include another example, note that the substitution `s/_[A-Za-z_]*_//g` eliminates everything tagged thus far.

Example: The following program shows how one can properly identify words in text. We shall refer to it as `leaveOnlyWords` in the sequel. (This is the longest program listing in this paper.)

```
#!/bin/sh
sed  ’s/[^A-Za-z.’\’’-][^A-Za-z.’\’’-]*/ /g
s/\([A-Za-z][A-Za-z]*\)\.\([A-Za-z][A-Za-z]*\)\./\1_\2_/g
s/\([A-Za-z][A-Za-z]*_[A-Za-z][A-Za-z]*\)\./\1_/g
s/Am\./Am_/g;    s/Ave\./Ave_/g;    s/Bart\./Bart_/g;
# (5)  The list of substitution commands continues  ...
s/vols\./vols_/g;    s/vs\./vs_/g;    s/wt\./wt_/g;
                                          s/\./ /g;      s/_/./g
s/\([A-Za-z]\)’\’’\([A-Za-z]\)/\1_\2/g;    s/’\’’/ /g;    s/_/’\’’/g
s/\([A-Za-z]\)\-\([A-Za-z]\)/\1_\2/g;      s/\-/ /g;      s/_/-/g’    $1
```

First, all strings which do not contain a letter, a period, an apostrophe or a hyphen are replaced by a blank (line 1). At this moment, the pattern space does not contain any

underscore character which is subsequently used as marker. Next (lines 2-3), strings of the type *letters.letters.* are replaced by *letters_letters_*. For example, `v.i.p.` is replaced by `v_i_p`. Following that, strings of the type *letters_letters.* are replaced by *letters_letters_*. For example, `v_i_p.` is then replaced by `v_i_p_`. Next (lines 4-6) comes a collection of substitution commands that replaces the period in standard abbreviations with an underscore character. Then (line 7), all period characters are replaced by blanks and subsequently all underscore characters by periods. Next (line 8), every apostrophe which is embedded in between two letters is replaced by an underscore character. All other apostrophes are then replaced by blanks, and subsequently all underscore characters are replaced by apostrophes. Finally (line 9), the hyphen is treated in a similar way as the apostrophe.

Example: The following program finds all words in a file `$1` that contain a double letter. We shall refer to this program as `doubleLetterWords`.

```
#!/bin/sh
leaveOnlyWords $1  |  addBlanks -  |\
sed  's/\([A-Za-z]\)\1/\1_\1/g;    s/ [^_ ][^_ ]* //g;    s/_//g'
```

In the first substitution command of the `sed` program, all double letters, which are encoded as `\([A-Za-z]\)\1`, are marked by a middle underscore character via `\1_\1`: e.g., `ll` is replaced by `l_l`. In the second substitution command, all unmarked words are deleted. To illustrate this by an example consider the following: after being processed by the first substitution command, the line
`Now, I will tell you why.`
looks like
` Now  I  wil_l  tel_l  you  why`
with a trailing blank. Consequently, the pattern in `s/ [^_ ][^_ ]* //g` cannot match `wil_l` and `tel_l` since no underscore character is permitted. Finally, the underscore characters are deleted.
Exercise: 1) Modify `doubleLetterWords` to search for double vowels as in `moon`.
2) Modify `doubleLetterWords` to search for pairs of consecutive vowels as in `beach`. Use only one tagged regular expression for the latter.
3) Modify `doubleLetterWords` to search for words ending in the string `ing` as in `swimming`. In that case, retain also the word that follows the word containing the string `ing`.
Application: This indicates how to select/mark/isolate not only words but also specific patterns in text.

Example: The following program replaces `@` by `@@`, `#` by `#@`, and `_` by `##` in a file, i.e., each of the single characters `@`, `#`, and `_` is replaced by the corresponding pair consisting of `@` and `#` only. In what follows, we shall refer to this program as `hideUnderscore`.

```
#!/bin/sh
sed  's/@/@@/g;    s/#/#@/g;    s/_/##/g'
```

The following program is the inverse of `hideUnderscore`, if both are seen as maps on files. In what follows, we shall refer to this inverse program as `restoreUnderscore`.

```
#!/bin/sh
sed  's/##/_/g;     s/#@/#/g;     s/@@/@/g'
```

First, `s/##/_/g` restores all underscores. Observe that `sed` scannes the pattern space from left to right. Consequently, the string `o####@` is correctly replaced by `o_#@`. Next, `s/#@/#/g` restores `#`. Finally, `s/@@/@/g` restores `@`.
Application: Being able to let a character (here the underscore) "disappear" in text at the beginning of a pipe is extremely useful. That character can be used to "break" complicated, general patterns for exceptions. This technique has already been demonstrated above in `leaveOnlyWords` and `doubleLetterWords`. Entities that have been recognized can also be marked by keywords of the sort `_DETERMINER_`. Framed by underscore characters, these keywords are easily distinguishable from regular words in the text. At the end of the pipe, all keywords are usually gone and the "missing" character is restored. Another application is to recognize the ending of sentences in the case of the period character. The period appears also in numbers and in abbreviations. By first replacing the period in the two latter cases by an underscore character and then interpreting the period as marker for the ending of sentences is, with minor additions, one way to generate a file which contains whole sentences per line.
(*Note:* Instead of using the triple (`@`, `#`, `_`) as above, one can use, in particular, (`7`, `8`, `9`). In that case, only the format of numbers changes occasionally. Usually, the format of numbers in text sources is not checked for the purpose of language analysis.)

4.9 A method for text analysis

At this point in our exposition, we can already formulate a simple method for text analysis. As outlined at the end of the last section, one has to implement the following steps:
1) If necessary, encrypt the source in such a way that one character which is unimportant for the subsequent analysis disappears from the text. This can be achieved by a program such as `hideUnderscore`.
2) Mark certain parts of the source that are either exceptional cases to the subsequent analysis or are cases of particular interest using the "hidden character" from 1) and, possibly, a collection of keywords. This has to be done in such a way that the pattern matching done under 3) does not apply to the special cases marked here.
3) Perform the required analysis or processing of the source "in general."
4) If necessary, invert the operation in 2) and, subsequently, the operation in 1).

4.10 Rearrangement of tagged regular expressions

We point out to the reader that the order of \1...\9 standing for tagged regular expressions need not be retained. Thus, rearrangement of tagged regular expressions is possible in the replacement in a substitution command.

Example: The following program acts on short sentences on single lines. For example, the sentence "Wilhelm is emperor." is replaced with "Which emperor was Wilhelm?"

```
#!/bin/sh
sed  's/^\([A-Za-z][A-Za-z]*\) is \([A-Za-z][A-Za-z]*\)\.$/Which \2 was \1?/'  $1
```

4.11 Line numbers as addresses

Besides the patterns defined above, one can use line numbers (without slashes) as addresses. Furthermore, the character $ stands for the last line.

Example: The following program copies the first 42 lines of a file $1 by quitting at line 42:

```
#!/bin/sh
sed  '42q'  $1
```

q is the command that causes sed to quit. Line 42 is the last line that is put into the pattern space and is processed (copied) by quitting. Consult man more and man less for alternatives to the above program.

Line numbers are cumulative over several files to which a sed program is applied. For example, the following two lines are the same:

```
sed '42q' file1 file2
cat file1 file2  |  sed '42q'
```

Example: The following program deletes the first and the last two lines in a file:

```
#!/bin/sh
sed '1d' $1  |  sed '$d'  |  sed '$d'
```

d is the sed command for deleting the current pattern space and starting a new cycle. Consult man tail and man less for alternatives to the above program.

The addresses 1 resp. $ can be used to insert headers resp. footers in documents using the commands i resp. a described below.

4.12 Address ranges

An address range has the format

Address1,*Address2*

It can be used in the same way as a single address when legal for a command. *Address1* specifies where (on which line resp. pattern space) actions begin. *Address2* specifies where actions end.

Example: Suppose that code is inserted in a `latex` document starting with a line containing up to white characters only `\BC` and ending with a line containing up to white characters only `\EC`. The following program indents the code by two blanks. In fact, non-empty code lines are indented only. All white ranges contain a blank and a tab.

```
#!/bin/sh
sed  '/^[  ]*\\BC[  ]*$/,/^[  ]*\\EC[  ]*$/s/./  &/
s/^[  ]*\\BC[  ]*$/\\begin{verbatim}/
s/^[  ]*\\EC[  ]*$/\\end{verbatim}/' $1
```

The program uses the address range

```
/^[  ]*\\BC[  ]*$/,/^[  ]*\\EC[  ]*$/
```

starting with the pattern `/^[  ]*\\BC[  ]*$/` and ending with `/^[  ]*\\EC[  ]*$/`. The period in the line addressed by the range matches only the first character in a non-empty pattern space since there is no `g` trailing the substitution command. At the end of the `sed` program, `\BC` resp. `\EC` are replaced by `\begin{verbatim}` resp. `\end{verbatim}`.

Example: The source code for this document contains several test programs for the claims made about `sed` commands in the next section. These programs are eliminated from the document through preprocessing with a one-line `sed` program. This is done in a similar fashion as above using a *begin* and an *end* address and the delete command `d` addressed by the range *begin*,*end*.

4.13 The list of all `sed` commands

Next, we include a list of all `sed` commands. In it, the number at the end of each section is the number of addresses possible. `2` means that address ranges are allowed. Usually, the command labeled by an address range is executed for every line in the range. We shall mention those commands that behave differently.

The most important commands are `b` for exclusion from processing, `d` for elimination of lines of input, `p` for additional printing during development of programs, `s` for global substitution, `t` for loops, `{}` for grouping and the not command `!` for negation of addresses. They are marked with two bullets • below. The others may be skipped on first reading.

• `a\`
Line1`\`

Line2\
...
LastLine

append: This prints *Line1* through *LastLine* at the end of the current cycle, i.e., after the content of the pattern space is processed with the `sed` program in the current cycle. Every line in the appended text except the last must end with a backslash `\`. What is appended is not put into the pattern space and not subject to the following `sed` commands. The appended text is printed even if the pattern space is deleted afterwards in the cycle or the quit command is executed. In connection with the last line address `$`, the `a` command can be used to append something to a file. `(1)`

•• `b` *whereTo*

branch: Branch to the `:` *whereTo* command where *whereTo* is a string of letters. If there is no *whereTo*, then branch to the end of the script. `b` is the classical "go to" command. The `:` *whereTo* may occur before `b` *whereTo* in the program creating a loop. In that case, another `b` command has to be used to leave the loop. Or, an address in front of the `b` command must deactivate the loop eventually. A `b` command without *whereTo* means "print the current pattern space and start processing the next cycle." With a `b` command without *whereTo* at the beginning of a `sed` program, certain lines that contain, e.g., a codeword which is matched by an address in front of `b` can be protected from being processed. `(2)`

Example: Suppose that one wants to manipulate the word "while" in a text that also contains listings of `C` code. Suppose, in addition, that the code is framed as in the example given above by `\BC` and `\EC`. In that case, a `sed` program starting

`sed '/^[ ]*\\BC[ ]*$/,/^[ ]*\\EC[ ]*$/b` would skip all code.

• `c`\
Line1\
...
LastLine

change: This prints *Line1* through *LastLine*. The current content of the pattern space is deleted, and a new cycle is started. Consequently, what is printed is not subject to the following `sed` commands. If an address range is given, then printing is done at the end of the address range. However, the current content of the pattern space is deleted for the full address range. Thus, with an address range one can exchange, e.g., a multi-line address or a paragraph in a document. `(2)`

•• `d`

delete all: The current content of the pattern space is deleted, and a new cycle is started. This can be used to remove all sorts of things including white lines via `/^[ ]*$/d`. `(2)`

• `D`

Delete initial segment: The initial segment of the pattern space through (including) the first newline character is deleted, and a new cycle is started with the remaining pattern space.

If the pattern space is empty, then a new line of input is processed, i.e., with no newline character in the pattern space `D` behaves as `d`. In that case, the first newline character is so to speak found and deleted at the end of the pattern space which makes the next line of input "visible." Something can be appended to the pattern space by the commands `G` and `N` described below. In those cases, a newline character separating old and new is appended first. `(2)`

Warning: The following program results in an endless loop:

```
sed 's/^/X\
/;D'
```

The added, initial segment `X`*newline* is deleted with `D`, and the original first line of input is reprocessed.

- `g`

get: Replace the contents of the pattern space by the contents of the hold space. If the hold space is empty, then this results in an empty pattern space. This is useful for repeated analysis of the original input line which can be stored in the hold space with the command `h`. `(2)`

- `G`

Get and append: Append the contents of the hold space to the pattern space. This includes appending a newline character first separating old and new. `(2)`

- `h`

hold: Replace the contents of the hold space by the contents of the pattern space. Storing the pattern space in the hold space makes it possible to reinvestigate the original line or an intermediate state of the pattern space. `(2)`

- `H`

Hold and append: Append the contents of the pattern space to the hold space. This includes appending a newline character first separating old and new. `(2)`

(*Note:* Both, `G` and `H` add newline characters while appending. Thus, an `H-G` sequence may create many empty lines due to double newline characters.)

- `i\`

text

insert: This prints *text* before the current content of the pattern space is processed and possibly printed with the `sed` program. As above for the `a` and `c` commands, every line in *text* but the last must end with a backslash `\`. What is inserted is not subject to the following `sed` commands. In connection with the first line address `1`, the `i` command can be used to prepend something to a document. `(1)`

- `l`

list: This lists the pattern space on the output in an unambiguous form. Non-printing characters are spelled in two digit ASCII and long lines are folded. This can be used to identify Japanese characters [Lunde 1993] in bilingual text. `(2)`

• `n`

next: This prints the pattern space. In addition, the next line of input is put into the pattern space. The current line number changes. However, a new cycle is not started from the top. Instead, the `sed` program is continued at the current program line for the pattern space with the new content. If there is no interference by other commands, then the switch by the `n` command in the pattern space is done for every second line of input. In the case of an address range, the addresses will only work, if the pattern space is matched *before* the `n` command is executed. Otherwise, the address is "overlooked." If the stop address is properly matched, then the effect of the `n` command reaches one line beyond the range. Compare the example given next. If `sed` is invoked as `sed -n`, then printing is suppressed, and only the next line of input is put into the pattern space. `(2)`

Example: The `n` command behaves as follows: `sed 's/a/b/g; /S/,/E/n; s/x/y/g'` yields from a file containing on separate lines the strings `1ax 2axS 3ax 4ax 5ax 6axE 7ax 8ax` the following sequence of strings (on separate lines): `1by 2bxS 3ay 4bx 5ay 6bxE 7ay 8by`. The lines 2, 4 and 6 were only subject to the first substitution command. The lines 3, 5 and 7 were only subject to the final substitution command. Note that `7ay` was obtained after `6bxE`. This shows that the `n` command may have consequences one line beyond an address range associated with it. The `8by` in the output shows that executing `n` stopped at `6bxE` since both substitution commands were applied to the line containing `8ax`. In contrast to that, `1ax 2axS 3ax 4ax 5axE 6ax 7ax 8ax` yields `1by 2bxS 3ay 4bx 5ayE 6bx 7ay 8bx`. This shows that the terminating address `/E/` is missed. In fact, `/5axE/` is put into the pattern space by the `n` command and can never be matched by the terminating address.

• `N`

Next is appended: This appends the next line of input to the pattern space with an embedded newline character separating old and new. The current line number changes. The newline character can be matched using `\n` and, e.g., be removed to unite lines. As above, `N` has an effect one line beyond a range and can miss an address, if the line matching the address is appended. If there is an attempt to append something beyond the the end of the file, then `sed` quits and misses processing and printing the last pattern space. `(2)`

•• `p`

print all: This prints the pattern space. Thus, in the usual `sed` mode one gets an additional line of output if the pattern space is not deleted afterwards. However, the default printing by `sed` can be switched off by invoking it as `sed -n`. `p` is very useful if one develops `sed` programs and prints intermediate stages of what is being processed. `(2)`

• `P`

Print initial segment: This prints the initial segment of the pattern space through the first newline character. `(2)`

•• `q`

quit: Print the current pattern space and terminate the `sed` program. `(1)`

- `r` *filename*

render: Copy the file *filename* to the output at the end of the current cycle. What is copied is not put into the pattern space. Copying is done even if the current pattern space is deleted or the `q` command is executed afterwards in the cycle. If no `n` or `N` commands are used, then the copying is done before processing the next input line. `(1)`

•• `s`/*pattern*/*replacement*/*flags*

substitute: Substitute *pattern* with *replacement*. *flags* is nothing or any intelligent combination of

n ($1 \leq n \leq 512$): Substitute only for the nth occurrence of the pattern.

`g`: Substitute globally for all non-overlapping occurrences of the pattern rather than just the first one.

`p`: Print the pattern space, if a substitution was made.

`w` *filename*: Append the pattern space to the file *filename*, if a substitution was made.

A legal `s` command is `s/old/new/512pw tFile` writing to file `tFile`. One can print to at most 10 different files. In case one has to use more files, one can split the `sed` program and use a pipe in which every piece uses up to 10 files. Larger text files that are processed may contain exceptional cases to patterns that are manipulated. Printing all performed substitutions to a shorter "diagnostic" file gives the user the opportunity for inspection. `(2)`

•• `t` *whereTo*

test: Branch to the label `:` *whereTo*, if any substitution has been made since the most recent reading of an input line or execution of a `t` command. If there is no *whereTo*, then start a new cycle. The `:` *whereTo* may occur before `t` *whereTo* in the program creating a loop. Actually, the `t` command is the one which is used regularly to create a loop since it becomes inactive when "nothing happened" prior to it. Creating loops with the `t` command can be used to (re)substitute in overlapping patterns. It can also be used for reprocessing if the pattern in a particular (preceeding) substitution is possibly generated by a subsequent substitution. `(2)`

- `w` *filename*:

write: Append the pattern space to the file *filename*. One can print to at most 10 different files. In case one has to use more files, one can split the `sed` program and use a pipe in which every piece uses up to 10 files. The `w` command can be used to sort pieces of a file into several files. Also, one can copy lines that match a certain pattern in a "diagnostic" file *before* a substitution is made. After a `w` command, everything that follows after some white space on the same line is understood as the filename to which the command is supposed to write. Thus, after a `w` command no other command can follow on the same line. `(2)`

- `x`

x-change: Exchange the pattern and the hold space. A sequence `x` *inspect* `x` may allow an intermediate inspection of the hold space containing a previous state of the pattern space

and connected with that, e.g., branching in the program. (2)

• y

yield: y/*string1*/*string2*/

Replace all occurrences of characters in *string1* with the corresponding character in *string2*. The lengths of *string1* and *string2* must be equal. Ranges are not allowed. A newline character in *string1* or *string2* can be represented just by typing ↩, i.e., introducing a newline character into the sed program without a preceeding backslash terminating the previous line. The slash / cannot occur in *string1* or *string2*. A substitution for it can be achieved by an additional s command. The y command can, e.g., be used to map lower-case to upper-case letters, and for elementary cryptography [Koblitz 1994]. (2)

• : *whereTo*

label: Address for the b *whereTo* and t *whereTo* commands. *whereTo* can be up to 7 characters long. (0)

• =

this is: Print the current line number of the input file on a separate line. As with the p command, printing is done immediately. (1)

•• {
commands
}

parentheses: Execute *commands* as a group, if the pattern space is matched by the address pattern preceeding {. Commands can be on separate lines or be separated by semicolons ;. Using a framing pair of parentheses, a non-address range command such as i can be applied to a range. (2)

Example: The following is legal: sed '/S/,/E/{p;s/^/a/;}'. Note the semicolon terminating the s command.

•• !*function*

not: Do not execute *function* if the pattern space matches the address pattern preceeding !. *function* can be a single command or a group specified by { and }. An address range is only allowed if *function* allows it.

(*addresses* the same number as for *function*)

4.14 Additional comments in regard to commands

Typically, the a, c, i and r commands are used if something should be maintained only at one place in some file. For example, a header containing an address may be inserted in a document several times. Or, a certain piece of code such as the declaration of a standard set of variables is used in many function definitions. The addresses 1 resp. $ can be used to insert headers resp. footers in documents using the commands i resp. a. This should be done only if the headers resp. footers are small, and the insertion is not always made. If a larger header or footer is not always added to a document, then using the r command

together with a separate file is more appropriate. If a header or footer is always added to a document, then using the UNIX command `cat` mentioned above together with separate files that contain the additions is best.

4.15 Additional examples

Example: The following program is another useful variation of `addBlanks`. It isolates non-white strings of characters in a text and puts every such string on a separate line. All ranges contain a blank and a tab. We shall call this `oneItemPerLine` in the sequel.

```
#!/bin/sh
sed  '/^[  ]*$/d;    s/^[  ]*//;    s/[  ]*$//;    s/[  ][  ]*/\
/g'  $1
```

First, all white lines are removed by deleting the pattern space which includes terminating the cycle. For non-white lines, white characters at the beginning and the end of lines are removed. Finally, all remaining strings of white characters are replaced by newline characters.

Example: The following program finds all four-letter-words in a text. We shall refer to it as `findFourLetterWords` in the sequel.

```
#!/bin/sh
leaveOnlyWords $1  |  addBlanks -  |\
sed  's/ \([A-Za-z][a-z][a-z][a-z]\) /_\1/g;  s/ [^_][^_]* //g;
      /^$/d;  s/_/ /g;  ='   |   sed  'N;  s/\n//'
```

The first `sed` program acts as follows: 1) All four-letter-words are marked with a leading underscore character. 2) All unmarked words are deleted. 3) Resulting white pattern spaces (lines) are deleted which also means that the cycle is interrupted and neither the line nor the corresponding line number are subsequently printed. 4) Underscore characters in the pattern space are replaced by blanks. 5) The line number is printed before the pattern space is. This will occur only if a four-letter-word was found on a line. The second `sed` program merges corresponding numbers and lines: 1) Every second line coming from the original source file `$1` which contains at least one four-letter-word is appended via `N` to the preceeding line containing solely the corresponding line number. 2) The embedded newline character is removed and the two united lines are printed as one.

Example: The following program sorts all characters `0` (zero) to the right of a line. This shows the typical use of the `t` command.

```
#!/bin/sh
sed  ': again;    s/0\([^0]\)/\10/g;    t again'  $1
```

The first command of the `sed` program defines the address `again`. The second command exchanges all characters `0` with a neighboring non-zero to the right. Hereby, the non-zero is

encoded as `[^0]`, tagged, and reused as `\1` in the replacement in the substitution command. The last command tests whether or not a substitution happened. If a substitution happened, then the cycle is continued at `: again`. Otherwise, the cycle is terminated.
Application: In the course of the investigation in [Abramson et al. 1995], [Abramson et al. 1996a], and [Abramson et al. 1996b], the file containing the source file of [Nelson 1962] (which was generated with a What-You-See-Is-What-You-Get Editor) was transformed by one of the authors into a `prolog` database. This raised the following problems:
1) The source is "dirty": it contains many control sequences coming from the wysiwyg-editor which have no meaning but the format and the spacing in the printed book. Such control sequences had to be removed. This was done using substitution commands with empty replacements.
2) The source cannot be "cleaned" in an easy fashion from the control sequences mentioned in 1). Some of the control sequences in the source are important in regard to the database which was generated. In [Nelson 1962], Japanese is represented using kanji, KUN pronunciation and *on* pronunciation. The *on* pronunciation of kanji is typeset in *italics* characters. In the source file, the associated text is framed by a unique pair of control sequences. Similarly, the KUN pronunciation of kanji is represented by SMALL CAPS printing.
3) The source was typed by a human with a regular layout on paper (i.e., in the printed book) in mind. Though quite regular already, it contains a certain collection of describable irregularities. For example, the ranges of framing pairs of control sequences overlap sometimes.
In order to match KUN pronunciation and *on* pronunciation in the source file of [Nelson 1962] properly, a collection of commutation rules for control sequences was implemented to achieve that the control sequences needed for pattern matching only frame a piece of text and no other control sequences. These commutation rules were implemented in a similar way as the last example shows.

Example: Suppose a file `$1` has the format of output of `leaveOnlyWords | addBlanks`, i.e., words on lines with proper spacing. The following program finds all words in a file `$1` that appear as first word and at least four times in a line. We shall refer to this program as `quadrupleWords`.

```
#!/bin/sh
sed  ': AD
      /^[^_]*___/!s/^ *\([A-Za-z'\''-][A-Za-z'\''-]*\)\([ _].*\) \1 /\1_\2/
      t AD
      /^[^_]*___/!d; s/^\([^_]*\)_.*$/\1/'  $1
```

The first command of the `sed` program defines the address `AD`. In the second command, a word at the beginning of a line, which occurs at least twice in the pattern space, is encoded as `[A-Za-z'\''-][A-Za-z'\''-]*`, tagged and repeated as `\1` in the pattern in the substitution command. In the replacement `\1_\2` in the the substitution command,

the first copy of the word is retained, a counting underscore character is appended to it, and the second copy (the first `\1`) is removed from the pattern space. The third command tests whether or not a substitution happened. If a substitution happened, then the cycle is continued at `: AD`. This is repeated at most three times since `/^[^_]*/___/!` preceeds the substitution command in the loop. If no substitution happened, then the cycle is continued in the next line of the program. In the last line, all pattern spaces are deleted that do not contain a triple underscore character corresponding to a quadruple word in the original line of input. In the last command of the program, everything after the first word is deleted.
Application: A procedure of identifying multiple patterns in longer sequences of words can be used to find words or patterns that are "locally" repeated and are (cf. [Hoey 1991, pp. 35-48, pp. 231-235]) very likely to be significant for the understanding of text. This will be outlined below in greater detail.

Example: The following program sorts all words in a text file `$1=fName` into several files depending upon the vowels occurring in the words. For example, all words containing the vowel "a" are put into one file `fName.a`. We shall refer to it as `sortByVowel` in the sequel.

```
#!/bin/sh
echo >$1.a;  echo >$1.e;  echo >$1.i;  echo >$1.o;  echo >$1.u;
leaveOnlyWords $1  |  oneItemPerLine -  |\
sed -n  '/a/w '$1'.a
         /e/w '$1'.e
         /i/w '$1'.i
         /o/w '$1'.o
         /u/w '$1'.u'
```

The second line of this `sh`-program generates empty files `$1.a`...`$1.u` in case the program has been used before on the same file. An alternative is to use the UNIX `rm`-command. Consult `man rm` for more details. Observe the use of the single quotes. If the argument `$1` to `sh` equals the string `fName`, then `sh` passes the following string to `sed` in regard to the first line of the `sed`-program: `/a/w fName.a`. `sed` then writes to the file `fName.a`. Note that output by the `w` command is always appended to an existing file. Thus, the files have to be removed or empty versions have to be created in case the program has been used before on the same file. There is no direct output by this UNIX command. It is clear how to generalize this procedure to a more significant analysis, e.g., searches for specific patterns or searches for phrases. Recall that everything after a `w` command and separating white space until the end of the line is understood as the filename the `w` command is supposed to write to.

Example: We shall refer to the following program as `mapToLowerCase`. It does what its name says.

```
#!/bin/sh
sed  'y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/'  $1
```

As outlined below, we have implemented a procedure in [Schmitt & Christianson 1998] which reformats text sources in such a way that whole sentences are on single lines. The latter procedure was applied to short essays submitted by Japanese students via e-mail as homework. We were subsequently interested in selecting student-generated example sentences containing a specific problematical pattern for presentation in class. The next two examples show such selection procedures. Similar programs can also be used to document the decline/increase of usage of a certain pattern over time.

Example: In teaching English as a second language, it is a common problem that students start many sentences with the word "I." If a file `$1` contains only sentences per line, then the following program will search for sentences that start with the word "I." We shall refer to it as `identifyBeginI`.

```
#!/bin/sh
sed  '/^[^A-Za-z]*I[^A-Za-z.]/!d'  $1
```

Note that `sed -n '/^[^A-Za-z]*I[^A-Za-z.]/p'` would work as well.
Also consult `man grep`. The `grep`-family of filters is designed to find lines in a file that match a certain pattern.

Example: The word "because" is invariably used incorrectly by Japanese learners of English. Because "because" is often used by Japanese learners of English to begin sentences (or sentence fragments), it is necessary to not only print sentences containing the string `Because` or `because`, but also to locate and print the immediately preceding sentence as well. The following program prints all lines in a file that match the pattern `/[Bb]ecause/` as well as the lines that preceed such lines. We shall refer to it as `printPredecessorBecause` in the sequel. Also consult `man grep` in regard to the options `-`n (n a positive integer), `-A`, `-B`, and `-C`.

```
#!/bin/sh
sed -n  '/[Bb]ecause/{x;p;g;p;b};     h'  $1
```

If the current line matches `/[Bb]ecause/`, then it is exchanged by the `x` command with its predecessor which was previously saved in the hold space. Next, the new pattern space containing the previous line is printed by `p`. Then, the pattern space is overwritten by `g` with the current line which is also printed by `p`. The `b` command terminates the cycle. If the current line does not match `/[Bb]ecause/`, then it is saved in the hold space through `h`.
Exercise: Write a `sed` program that prints a line matching `/[Bb]ecause/` and the two lines preceeding it. Write an `awk` program that does the same as the latter program.

Exercise: Implement the `awk` program `double` given in [Kernighan & Pike 1984, p. 121] using `sed` and the pipe mechanism. Disregard applying the program to multiple files. Print the filename using `echo`. Use tagged regular expressions in order to recognize the double words. Use appending lines with the `N` command to the pattern space in order to recognize

double words that are spread over the newline character that separates two lines. Be aware of properly processing the last line in connection with the `N` command. If you want to make this exercise difficult, then study [Kernighan & Pike 1984] in regard to `for` loops with `sh` and `sh` variables and implement processing several files through a `for` loop in `sh` over the filenames using the `sed` program that was just developed.

4.16 Generating a search program from a file with data

One can use a `sed` program to create another program from a file containing data in a convenient format, e.g., a list of words. Such action can preceed the use of the generated program. I.e., one invokes the two programs separately. Alternatively, the generation of a program and its subsequent use are part of a single UNIX command. The latter possibility is outlined next.

Example: Suppose that one has a file that contains a list of words that are "unimportant" for some reason, and that one wants to eliminate them from a second text file. For example, the words *the*, *a*, *an*, *if*, *then*, *and*, *or*, ... may be unimportant, if one wants to define, e.g., "context." See [Butler 1985, pp. 219-220] [Woods 1995] for lists of frequent words. The following program generates a `sed` program `$1.sed` out of a file `$1` that is supposed to contain a list of "unimportant words". The generated script `$1.sed` eliminates the words in `$1` from a second file whose name is given to `sh` as second argument `$2`. We shall refer to the following program as `eliminateList` in the sequel.

```
#!/bin/sh
leaveOnlyWords $1  |  oneItemPerLine -  |\
sed  's/[./-]/\\&/g
      s/.*/s\/\\([^A-Za-z]\\)&\\([^A-Za-z]\\)\/\\1\\2\/g/' >$1.sed
addBlanks $2  |  sed -f $1.sed  |  adjustBlankTabs -
```

The second line in the program isolates words in the file `$1` and feeds them (one word per line) into the subsequent first `sed` program. In the first `sed` program in lines $3-4$ the following is done: 1) Periods, slashes ("A/C"), or hyphens are preceeded by a backslash character. For example, the string `built-in` is replaced by `built\-in`. This is done since periods and hyphens are special characters in `sed`. 2) The second substitution command generates an `s` command from a given string on a single line. In fact, out of a line containing solely the string `built\-in` which is matched by `.*` and reproduced by `&` in the second substitution command listed above the following `s` command is generated in `$1.sed`:

```
s/\([^A-Za-z]\)built\-in([^A-Za-z]\)/\1\2/g
```

Note that all slash and backslash characters occurring in the latter line (except the one in `built\-in`) have to be preceeded by an additional backslash in the replacement

```
s\/\\([^A-Za-z]\\)&\\([^A-Za-z]\\)\/\\1\\2\/g
```

in the generating second substitution command listed above. The list of so generated `s`

commands is stored in `$1.sed`. Using `sed -f $1.sed`, this file of `s` commands is then applied to the file whose name is given to `sh` as second argument `$2`.

Example: In [Schmitt & Christianson 1998], we use several filters (i.e., `sed` programs) that are generated in the same way as the last example. Characteristic in each case is that we generate these programs during the setup of the system used in [Schmitt & Christianson 1998]. This gives the user the ability to customize his/her working environment that is generated by our system. For example, we have implemented a filter `hideAbbreviations` and its left inverse filter. `hideAbbreviations` has two parts (compare the listing of `leaveOnlyWords` given above). The first recognizes abbreviations of the sort `v.i.p.` and replaces the periods by underscore characters. The second part is generated from a list of strings containing, e.g., `Am`. Out of those, replacement commands are generated that replace, e.g., `Am.` by `Am_`. This filter is used in the program that reformats essays such that lines contain whole sentences. The latter procedure will be explained below in greater detail.

4.17 Elementary grammatical analysis and its applications to teaching

Similar to the last example, one can generate programs such as `markDeterminers` from a source file containing only a list of words. Different types of words can be tagged or replaced by their grammatical type by automatically generated `sed` programs. For example, one could have a file `verbs` containing (selected) verbs and generate a program marking those verbs in text. If one processes larger files, then, possibly, one should generate `C` programs based upon `lex` to perform searching and tagging. Note that a `C` program P_1, in which a list of words to search for is encoded, is as fast as a `C` program P_2, that has to read a list of words it is supposed to identify. Also, tagging can be done in regard to specialized word lists such as [Orr et al. 1995] or general word lists such as the General Service List [Nation 1990]. A collection of such search programs can be used for analysis of grammatical patterns in texts involving selected verbs, nouns and other components.

By tagging a given word list, the foreign language teacher is able to do searches for grammatical trouble spots. Numerous tagging schemes are currently in use in large-scale corpora (cf., [Greenbaum 1991], [Leech & Fligelstone 1992], and [Meyer & Tenney 1993]). Most of these are extremely detailed schemes used to explore corpora consisting of tens of millions of words. Corpus linguists involved in such research require very high accuracy. However, the average foreign language teacher, working with a word list of probably 1000-5000 words (likely built up over time) requires a great deal less sophistication. A rough and very general tagging scheme like the one shown in `markDeterminers` is enough for most practical applications in which the human end-user can correct a small number of exceptional cases.

Armed with a tagged word list and a few simple `sed` and `awk` programs, a teacher could search, for example, for every occurrence of *had+[past participle]* to check for certain mistakes in past perfect constructions or the quantifiers *every/many/some/all/...* to check for

mistakes in pluralization.

The programs `countFrequencies`, which counts the number of occurrences of "items" and is fully explained below, can be employed in order to gather statistics of patterns used. Such statistics are useful for a language teacher in determining which patterns students feel more secure about using (i.e., use more often) and those not used for one reason or another. Avoidance is a difficult aspect of language use to measure. However, using a program which analyzes sentence patterns, prints like patterns in files, and keeps statistics regarding frequency of use, patterns which students rarely use would be immediately apparent to the teacher. Searching for grammatical patterns can also be used to select example sentences from a database of, e.g., homework that was actually written by students for presentation by the instructor in class (cf. [Schmitt & Christianson 1998]). Using the set and vector operations defined below, patterns that are used can be measured against patterns that are desirable and were introduced in class. The system in [Schmitt & Christianson 1998] is designed in such a way that search programs of this kind can easily be incorporated.

Also, it is easy to alter the above idea to generate out of a file with lines in the format

```
deutsch { english1 english2 ... }
```

containing the translations for the (say) 2000 most common German words a `C` program that copies sentences on lines as `latex` comment and delivers an associated local dictionary.

5 awk

`awk` is a simple programming language based on pattern recognition and operations on chunks of the input record. Usually, an input record is an input line. The chunks of the input record are called fields and, usually, are the full strings of non-white characters. In contrast to `sed`, `awk` allows string variables and numerical variables. Consequently, one can accomplish operations on files such as accounting and keeping statistics of things. Another typical use of `awk` is matching and rearranging the fields in a line. Good introductions to `awk` are [Aho et al. 1978] and [Aho et al. 1988].

In what follows, we shall sometimes include an `awk` version of a procedure implemented above with `sed`. This allows adaptation of the procedures (such as inclusion into other `sed` or `awk` programs) under different circumstances.

5.1 `awk` programs

Programs in `awk` need no compilation. An `awk` program looks as follows:

```
awk 'BEGIN    { actionB }
pattern1      { action1 }
```

pattern2 { *action2* }
...
END { *actionE* }'

actionB is executed before the input is processed. *actionE* is executed after the input is processed. The lines with BEGIN and END can be omitted. Or an awk program can consist of, e.g., an END-line only. As shown above, any awk command or awk statement has the following format:

pattern { *action* } ;

The closing semicolon is optional. If a semicolon follows an awk statement, then another statement can follow on the same line. The statements with the BEGIN and the END pattern must be on separate lines. One can also store a list of awk commands in a file (say) awkCommands and use awk -f awkCommands to invoke the program.

patterns can be very similar to address patterns in sed. However, more complicated address patterns are also possible. Compare the two listings given below.

The input record is put into a pattern space as in sed. However in awk, the pattern space is divided into an array of the fields of the original input record. Each of these fields can be manipulated separately. Since the whole input record can be stored as a string in any variable, awk does not need a hold space.

awk operates on input records (usually lines) in a cycle just like sed. *action1* is executed if *pattern1* matches the original input record. After that, *action2* is executed if *pattern2* matches the current, possibly altered pattern space and the cycle was not terminated by *action1*. And so on. An action is a sequence of statements (commands) that are separated by semicolons ; or are on different lines. If *pattern* is omitted, then the corresponding action is done all the time provided this program line is reached in the cycle. If { *action* } is omitted, then the whole input line is printed by default. Observe that by default an awk program does *not* copy an input line (similar to sed -n). Thus, printing has to be triggered by an address pattern, which selects the pattern space as shown in the next example, or printing has to be triggered by a separate print statement.

Example: The following program does the same as identifyBeginI. The default action is used, i.e., to print a line that matches the given address pattern.

```
#!/bin/sh
awk  '/^[^A-Za-z]*I[^A-Za-z.]/'  $1
```

5.2 Patterns I (regular expressions)

Patterns are used in awk as address patterns to select the pattern space for an action. They can also be used in the if statement of awk to define a conditional. In this section, we shall discuss those patterns in awk which are called regular expressions. Regular expressions in awk

are very similar to regular expressions in `lex` and `sed`. In addition to the list of patterns which we give next, there is also the possibility to define arithmetic-relational expressions, string-valued expressions and arbitrary Boolean combinations of all of the above. The patterns different from regular expressions will be explained later.

A regular expression is framed by a leading and a terminating slash `/`. The following rules must be observed:

1) Any non-special character matches itself.

2) Special characters that otherwise have a particular function in `awk` have to be preceeded by a backslash `\` in order to be understood literally. A newline character in the pattern space can be matched with `\n`.

All special characters: `\\ \/ \^ \$ \. \[ \] \* \+ \? \( \) \| \n`

Example: `awk '/\(\?\)/'` prints lines that contain the string `(?)`.

(*Note:* The strings `\(` and `\)` are not used for tagging. There is no tagging in `awk`. The ampersand `&` is not special in `awk`.)

3) `^` resp. `$` match the beginning resp. the end of the pattern space.

Example: `awk '/^/'` prints every line of input.

4) `.` matches any single character.

Example: `awk '/./'` prints every non-empty line of input.

`awk '/....................../'` prints every line that contains more than 20 characters.

5) `[`*what*`]` matches any character in *what* where *what* is a string of characters.

R1: The backslash `\` is not needed and used to indicate most special characters in *what*. One exception is `\]` which represents `]`. The backslash itself should always be represented as `\\` in *what* even though sometimes one backslash in the middle of a longer *what* is enough.

R2: Ranges of the type `a-z`, `A-Z`, `0-9` in *what* are permitted.

R3: The hyphen `-` must be at the beginning or the end of *what* in order to be recognized as itself.

R4: The carat `^` must not be the first character in *what* in order to be recognized as itself.

6) `[^`*what*`]` matches any character not in *what*. The rules **R1**–**R3** set under 5) also apply here.

7) The ordinary parenthesis `(` and `)` are used for grouping.

Example: `/0.(07)*\$/` matches `0.$`, `0.07$`, `0.0707$` etc. but not `0.0$`. Observe the radical difference from the pattern `/0.0*7*\$/`.

8) The vertical slash `|` is used to define alternatives.

Examples: `/new|old democrat/` matches "`new`" and "`old democrat`." Observe that this is different from `/(new|old) democrat/` which matches `new democrat` and `old democrat`.

9) *pattern*`repitor` stands for 0 or any number of copies of *pattern*, if `repitor=*`. It stands for any strictly positive number of copies of *pattern*, if `repitor=+`. It stands for 0 or 1 copy of *pattern*, if `repitor=?`. *pattern* is a specific character, the period `.` meaning any character, a range `[...]` as described under 5) or 6) or something in parenthesis.

Examples: `awk '/^[a-zA-Z]+$/'` prints all non-empty lines that contain solely letters. `awk '/baka(geta)? desu/'` prints all lines containing `baka desu` or `bakageta desu`.

5.3 Representation of strings

Strings of characters in `awk` are framed by double quotes `"`. The sequences `\"`, `\t` resp. `\n` represent the double quote, the tab resp. the newline character in strings. The backslash can sometimes by included in strings as a single backslash character. However, as a rule one should always use the sequence `\\` to represent the backslash. Otherwise, every character including the blank just represents itself.
Example: `"\"The Lying King\""` represents the string `"The Lying King"` including the double quotes.
(*Note:* `man awk` may list "There is no escape sequence that prints a double-quote." as a bug of `awk`. However, the string `"\""` can be printed without problems using the statements `print` or `printf` described below.)

Strings can be concatenated by just writing them behind each other separated by blanks. This hold also for variables containing strings.

5.4 Fields and field variables

In the default mode, the fields of an input line are the full strings of non-white characters separated by blanks and tabs. Thus, in

```
Errare humanum                    est.
```

there are three fields: `Errare`, `humanum` and `est.` (the last field includes the period). They are addressed in the pattern space from left to right as field variables `$1`, `$2` and `$3`. Alternatively, they can be addresses as `$(1)`, `$(2)` and `$(3)`. One can count beyond 9 in regard to field variables. `$0` and `$(0)` stand for the whole pattern space.

5.5 Printing using print

Usually, one uses the `print` statement for printing as in the next example. The program

```
#!/bin/sh
awk  '/rare?/ {  print $3 $2 , $1 "???"  }'  $1
```

prints `est.humanum Errare???` from the line given above. The `print` statement prints the field variables `$3`...`$1` and, finally, the string `"???"`. The comma separating the field variables `$2` and `$1` in the `print` statement causes a blank to be printed. In our example, that is the blank in `humanum Errare`.

In general, a `print` statement has the following structure:

`print` *object*`separator`*object*`...separator`*object*

object can be either a string of characters framed by double quotes, a number or a variable name. `separator` is a string of characters containing solely blanks and at most one comma. A comma causes an output field separator string `OFS` (default a blank) to be printed. The sequence of arguments may either be empty or must end in *object*. The `print` statement is terminated by a greater sign `>`, a semicolon `;`, a closing brace `}`, or the end of the line. After a `print` statement, an output record separator `ORS` (default a newline character) is printed. We shall show below how the variables `OFS` and `ORS` can be reset.

The action `{ print }` is as good as `{ print $0 }` for printing the pattern space. If nothing is done with the pattern space and there is a selecting address, then `{ print }` as action can be omitted completely.

Note that `print money` is interpreted as printing the *content* of the variable `money` while `print "money"` really prints `money`. The action `{ print "" }` prints (the empty string plus) an output record separator `ORS`. Finally, note the very different meanings of `$1` (first field of line vs. first argument to shell) inside and outside the `awk` program in the example.

Example: The following program prints the first five fields in every line separated by one blank. It can be used to isolate starting phrases of sentences.

```
#!/bin/sh
awk  '{  print $1 , $2 , $3 , $4 , $5  }'  $1
```

Example: The following program triple spaces the input file. This can be useful if one wants to correct printed text by hand and needs space for inserted comments.

```
#!/bin/sh
awk  '{  print $0 "\n\n"  }'  $1
```

The `print` statement prints the input line contained in `$0` concatenated with two following newline characters and a third newline character as `ORS`.

Example: The following program prepends the second field in a line by a newline character if there are at least two fields in the line, and then prints the pattern space (possibly printing two lines).

```
#!/bin/sh
awk  '{  $2="\n" $2 ;   print  }'  $1
```

`=` is the assignment operator in `awk` working from left to right. In this example, the second field `$2` (left) is assigned the string that results from concatenating a newline character with the content of `$2` (right). This is done only if a second field exists. I.e., a second field `$2` is not initiated to the empty string and then united with the newline character.

5.6 Formatted printing using printf

This section may be skipped, if the reader is not interested in printing tabular output containing numbers.

The second function for printing in `awk` is `printf`. Let us implement another version of the example which printed `est.humanum Errare???` using `printf`:

```
#!/bin/sh
awk  '/rare?/ {  printf "%s%s %s???\n", $3, $2, $1  }'  $1
```

The `printf` statement prints the field variables `$3`...`$1` which are listed as its arguments in `, $3, $2, $1` at the end. How these variables are printed is specified in the *format* string `"%s%s %s???\n"`. It says: Print the first variable listed as argument as a string (`%s`) immediately followed by the second variable also understood as string. Then print a blank followed by the third variable understood as string. Finally, print the string `"???\n"` including a newline character.

In general, a `printf` statement has the following structure:

`printf` *format* , *expression1* , *expression2 ...*

format follows the rules for formatting of `printf` from the programming language `C`. Basically, the *format* string in a string of characters (some preceeded by the escape character backslash \) in which specifications starting in `%` are embedded. The most important specifications are:

- `%c` says that the content of a variable is supposed to be an integer n satisfying $32 \leq n \leq 126$ and the corresponding ASCII character is printed.

Example: `awk '{ for (k=32;k<=126;k++) { printf "(%d:%c)\n",k,k}}'` prints a list after a ↩. The `for` loop will be explained below.

- `%`n`d` prints an integer in a field which is n characters wide with leading blanks.

Example: `20` is printed by `%14d` as `20` including 12 leading blanks.

- `%`n`.`m`e` prints a number in "scientific" format in a field which is n characters wide with leading blanks using a mantissa which has one digit in front and m digits after the period.

Example: `43` is printed by `%12.3e` as `4.300e+01` including 3 leading blanks.

- `%`n`.`m`f` prints a floating point number in a field which is n characters wide with leading blanks and m digits after the period.

Example: `552` is printed by `%10.4f` as `552.0000` including 2 leading blanks.

- `%s` says that the content of a variable is supposed to be printed as string. In particular, a number from the input record contained in a field variable is printed unchanged.
- Numbers can also be printed signed or in octal and hexadecimal representation. In addition, they can be printed starting left in the field. It can be specified that zeroes rather than blanks fill a field in which a number is printed. This is described in detail in `man 3 printf` and [Kernighan & Ritchie 1988, Appendix B, p. 243].

The `printf` statement is terminated by a greater sign `>`, a semicolon `;`, a closing brace `}`, or the end of the line. `printf` does not print a terminating output record separator. Newline characters have to be explicitly included in *format* as `\n`.

Example: The following `sh` program shows the use of `printf`:

```
#!/bin/sh
awk '{printf "Field $1:%14d; $2 pl $3:%12.3e; $4 ti $5:%10.4f.\%\"\n",$1,$2+$3,$4*$5}'
```

It produces from a line containing `20 21 22 23 24` the following:

```
Field $1:             20; $2 pl $3:   4.300e+01; $4 ti $5:  552.0000.\%"
```

Essentially, the text in the *format* is reproduced. In particular, the starting substring `Field $1:` is just reproduced as string and the substring `$1` in it is not interpreted as field variable. The sum `$2+$3` and the product `$4*$5` at the end of the line in the program initiate the corresponding computations whose results are printed. Specifying formats in the way of the present example is useful to obtain nice tabular output of statistical evidence and other accounting. For the representation of special characters consult the discussion of strings in `awk` given above.

We leave the discussion of `printf` at this point since it is more important in regard to numerical computation with `awk` rather than to text processing. In regard to the latter, `print` is mostly sufficient in our experience.

5.7 Printing to files and pipes

`>>` can be used within an `awk` program to append output to a possibly existing file whose name has to be included in the program. An output file is created if it is not in existence. Using a single `>` instead overwrites an existing file and only appends output from the currently running `awk` program to the file after that.

One can print to at most 10 different files from within an `awk` program. If one needs to print to more files, then one can split the program into several pieces and use a pipe.

Application: Printing to different files can be used to sort things or to generate "diagnostic" files for inspection by the human user of lines that match special patterns.

Example: The following `sh` program interprets the first field `$1` in a line of the input file as a file name to which a line containing the second field `$2` is appended. The second field is supposed to be a non-zero integer.

```
#!/bin/sh
awk  '/^[  ]*[A-Za-z]+[  ]+[1-9][0-9]*[  ]*$/ { print $2 >> $1 }'  $1
```

We check in the address pattern 1) that there are only two fields, 2) that the first field consists of letters only, and 3) that the second field is a non-zero integer. The white ranges contain a blank and a tab each. Note the `+` behind the second white range assuring white

space separating $1 and $2. The output of print $2 is appended via >> to the file whose name is contained in the first field $1.

Example: The following program mimmicks the UNIX command tee. The input file is once copied line-by-line to the output by the first print statement. In addition, every line is copied to a separate file $2 whose name is argument to the sh program.

```
#!/bin/sh
awk  '{  print ;  print >>"'$2'"  }'  $1
```

Observe the use of the single quotes. If the second argument $2 to the sh program is fname, then sh delivers the substring { print ; print >>"fname"} to awk. fname must be presented to awk within double quotes since otherwise it would be understood as a variable name. Again, this shows a simple technique to place arguments to sh commands into sed or awk programs. tee is useful to collect "diagnostic" files of intermediate stages of a file that is processed in a pipe. Consult man tee for more details and options on tee.

Example: The following program is another version of sortByVowel.

```
#!/bin/sh
leaveOnlyWords $1  |  oneItemPerLine -  |\
awk   '/a/ { print >"'$1'.a" };    /e/ { print >"'$1'.e" }
       /i/ { print >"'$1'.i" };    /o/ { print >"'$1'.o" }
       /u/ { print >"'$1'.u" }'
```

Since we can use > within awk it is not necessary to erase the files $1.a...$1.u in a first step. The strings defining the files are not stored in variables. Thus, they have to be framed by double quotes ". Otherwise, the sed and awk versions of sortByVowel are very similar.

Instead of appending output to files, one can also feed output into a pipe:

```
#!/bin/sh
awk  '/MUSE/ {  print | "mail kanda"  }'  $1
```

This mails all lines in a file containing the address pattern MUSE to the user kanda (together in a single mail). Of course, awk '/MUSE/' $1 | mail kanda is a better version of the example in the spirit of UNIX. But from within awk one can sort and mail different things to several recipients at the same time. Consult man mail for more details about mail. It is useful to know how to mail entire files with /usr/ucb/mail.

5.8 Variables

In awk, variables do not need to be declared. They just exist and can be set to (filled with) strings and numbers of different types (i.e., integer, floating point or scientific format). Variables are initiated to the empty string automatically. The empty string is interpreted as 0 if a variable is used in a numerical computation.

Example: The following program exchanges the first two fields in every line of a file.

```
#!/bin/sh
awk  '{  s=$2;  $2=$1;  $1=s;  print  }'  $1
```

The second field `$2` is stored temporarily in the variable `s` via `s=$2`. Then, the content of `$2` is overwritten with the content of the first field `$1` through `$2=$1`. Finally, the original content of `$2` is put into `$1` via `$1=s` and the altered pattern space is printed.

Example: The following program is another version of `printPredecessorBecause`.

```
#!/bin/sh
awk  '/[Bb]ecause/ { print previous "\n" $0 };    { previous=$0 }'  $1
```

If the current line matches `/[Bb]ecause/`, then it is printed following its predecessor which was previously saved in the variable `previous`. Finally, every line is saved in the variable `previous` waiting for the next cycle.

5.9 Arrays (regular format)

Arrays simply exist after first use. Their dimension is 1 (meaning one index consisting of numbers). Their size, i.e. the number of elements, need not be declared. Any element of an array that is used is initiated to the empty string which is interpreted as 0 in numerical computations. An element of an array is denoted as `name[`*index*`]`. `name` is the name of the array and *index* is a number. Elements of arrays can hold everything that ordinary variables can, i.e., strings and different types of numbers.

5.10 Associative arrays

Associative arrays of the format `name[`*index*`]` are exactly like regular arrays except that *index* is a string. The empty string is allowed as *index*.

5.11 Built-in variables

Next, we include the list of built-in variables of `awk`. All built-in variables can be used in the same way as other variables in computations, string manipulations and conditionals. In particular, all built-in variables can be reset. Usually, a built-in variable is reset in the first group of statements *actionB* of an `awk` program addressed by the `BEGIN` pattern.

- `FILENAME`

`FILENAME` contains the name of the current input file. `awk` can distinguish the standard input as name of the current input file.
Example: `FILENAME=="-"` is a legal conditional in an `if` statement and a legal address pattern. This will be explained in more detail below.

- `FS`

FS contains the field separator character. The default are sequences of blanks and tabs. This is slightly beyond a single character but is done for convenience. Note that the assignment FS=" " causes the field separator character to be set to *sequences* of blanks and tabs.
Example: If one processes chunks of texts not words, then one may want to set the field separator to, e.g., # through FS="#". In that way, one can process phrases and sentences where words are separated by blanks while the fields are separated by #. One application is to separate original and translation of phrases by #. If the field separator is set to #, then n characters # in the line define $n + 1$ possibly empty fields, i.e., NF= $n + 1$.

- NF

NF contains the number of fields in the current pattern space (input record). This is very important in order to loop over all fields. Note that NF can be increased to "make room" for more fields which can be filled with results of the current computation in the cycle.

- NR

NR contains the number of the most recent input record. Usually, this is the line number if the record separator character RS is not reset or NR itself is not reassigned another value. Note that NR counts cumulatively over several files to which an awk program is applied. In that case, one may wish to test the variable FILENAME for change and reset NR=1 at the beginning of a new file. Consult [Kernighan & Pike 1984, p. 121] for such an example.

- OFMT

OFMT contains the output format for numbers used by print. The default is %.6g. Consult the above section on printf and man 3 printf for details about such formats.
Example: The assignment OFMT="%e" causes all numbers to be printed in "scientific format."
(*Note:* Primarily, a number is considered as a string as long as it was not subject to a computation. OFMT only becomes active for numbers that were involved in computations. If numbers are just copied with print, then they reappear unchanged from the input format. If a number is stored in variable x, then x=x+0 does the trick of activating OFMT. x=x is only an assignment of strings.)

- OFS

OFS contains the output field separator used in print. OFS is caused to be printed if a comma , is used in a print statement. The default is a blank character. It can only be one character long. In particular, it should not be set to the empty string.
Example: OFS="\n" sets OFS to the newline character.
(*Note:* OFS stays inactive, if the string $0 was never manipulated in the awk program and is then printed with the print statement. For example, if a pattern space has just been selected for printing through an address pattern, then it is printed unchanged provided no alteration to $0 has been made previously in the cycle. If one uses a dummy statement, e.g., $1=$1 which does nothing to the first field $1, then the decomposition of $0 into an array of field variables becomes active.)
(*Note:* It seems advisable to exchange field separators and to increase spacing with sed.)

- ORS

ORS contains the output record separator string. It is appended to the output after each print statement. The default is a newline character.
Examples: ORS can be set to the empty string through ORS="". This can be used to unite lines of the input file. If one so desires, then it can also be set to two newline characters via ORS="\n\n" double spacing the output.

- RS

RS contains the input record separator character. The default is a newline character.
Examples: If one sets FS="\n" and RS="", then the fields are whole lines and an input record is a paragraph limited by empty lines.

We shall give some examples for the use of FILENAME later in connection with set operations on files (the program setIntersection) and in connection with the implementation of a vocabulary trainer.

The built-in variable NF is mostly used in connection with the for statement. Check the section describing the for statement for some examples using NF. Other examples will be given in operations on lists of type/token ratios.

Example: The following sh program counts the number of paragraphs in a text file.

```
#!/bin/sh
sed  's/^[  ]*$//'  $1  |\
awk  'BEGIN  { FS="\n" ;  RS="" }
      /^$/   { NR=NR-1  }
      END    { print NR }'
```

The first sed program sets white lines in the input to empty lines. The idea in the awk program is to count and print the number of paragraphs which are put as one input records into the pattern space. As indicated above, the fields in this setting are the individual lines of text. Two newline characters in a row define an "empty" field which matches RS. In case there is a triple newline character, the two "empty" fields in a row create an empty pattern space matched by /^$/ but not a new paragraph. Thus, the number of input records which is increased by awk automatically has to be decreased by 1 using NR=NR-1 (or NR-=1). The corrected number of records NR which equals the number of paragraphs is printed at the end.

Examples: The following sh program is another version of oneItemPerLine. The range contains a blank and a tab.

```
#!/bin/sh
awk  'BEGIN  { OFS="\n" }
     /[^  ]/ { $1=$1 ;  print }'  $1
```

The statement $1=$1 manipulates the pattern space containing $0 for non-white lines. Then, $0 is printed through print field-wise using OFS=*newline*. Thus, all fields are on separate lines. The last field is separated from the next line through ORS=*newline* by default.

5.12 Definition: a list of type/token ratios

In this section, we define a very useful file format. Suppose that one represents frequencies of use/occurren- ce of particular words or phrases in the following way: First in every line comes a word or phrase which can contain a number. In addition to that, the final field of every line contains a number which may, e.g., count how often the preceeding entity occurred in a text file or may denote the relative probability of the preceeding entity. A file in this format will be called a *list of type/token ratios*. An example of an entry is given by

```
limit  55
```

The last entry will be called the *frequency* of the preceeding word or phrase. Mathematically speaking, such a file of word/phrase frequencies is a vector over the free base of character strings [Greub 1981, p. 13].

5.13 Operators

`awk` has built-in operators for numerical computation, Boolean or logical operations, string manipulation, pattern matching and assignment of values to variables. The following lists all `awk` operators in decreasing order of precedence, i.e., operators on top of this list are applied before operators that are listed subsequently, if the order of execution is not explicitly set by parenthesis.

Note that strings other than those that have the format of numbers all have the value 0 in a numerical computations.

- `++ --`

`++var` increments the variable `var` by 1 before it is used. `var++` increments `var` by 1 immediately after it was used (in that particular spot of the expression and the program). `--var` decrements `var` by 1 before it is used. `var--` decrements `var` by 1 immediately after it was used.

(*Note:* `++` and `--` are very useful for counting. In particular, they are used for counting in `for` and `while` loops.)

- `* / %`

Operations with numbers: multiplication, division, integer division remainder.

Example: The conditional `NR%3==1` yields *true* at record number 1, 4, 7 etc. Consequently, `awk 'NR%3==1'` prints every third line in a file.

- `+ -`

Operations with numbers: addition and subtraction.

- *nothing* or better separating blanks

Concatenation of strings.

Examples: Two strings `"aa"` and `"bb"` can be concatenated via `"aa""bb"` to `"aabb"`. The strings in two variables `x` and `y` can be concatenated and assigned to, e.g., a variable `z` via `z=x y`. Here, the blank is needed as a separator.

- `> >= < <= == != ~ !~`

Comparing expressions, in particular comparing numbers: greater, greater or equal, less, less or equal, equal, not equal, matches pattern, does not match pattern. The first six operators are regularly used to compare numbers. They can also be applied to strings. The last four operators are regularly used to compared strings. If it is not clear what sort of comparison is meant, then `awk` uses string comparison instead of numerical comparison.
(*Note:* In particular, the expression `$0`, which stands for the whole input record, is seen as a string. If this string has the format of a number, then `$1` picks the number out of the string so to speak. With the operators `~` (matches) `!~` (matches not), one can, e.g., test variables and, in particular, fields against patterns.)

- `!`

Logical not. The logical not has to preceed an expression. In particular, it has to preceed an address in order to negate it. (This is different from the notation of the "not" command `!` in `sed`.)
Example: `awk '!/^.?(..)?(....)?(........)?(.....)?$/'` prints every line with more than 20 characters.

- `&&`

Logical and.

- `||`

Logical or. (*Note:* Keep in mind that the logical or is not exclusive, i.e., *true*`||`*true* yields *true*.)

- `= += -= *= /= %=`

`=` is the assignment operator. `var=result` sets `var` to (the content of) `result`. The other assignment operators exist just for notational convenience. `var+=d` sets `var` to `var+d`. The statement `var+=d` is exactly the same as the statement `var=var+d`. `var-=d` sets `var` to `var-d`. Etc.

The list given above defines the order of precedence in algebraic expressions. To illustrate, suppose that `x` is a variable that holds the number 2. Then `y=++x*x` sets `x` to 3 before it is used because the sequence `++` is to the left of `x`. Consequently, `y` is set to 9. On the other hand, `y=x++*x` sets `x` to 3 after the variable is used for the first time because the sequence `++` is to the right of the first `x`. Since `awk` scans algebraic expressions from left to right and multiplication has lower precedence than `++`, the variable is increased before the second `x` and the product are evaluated. Consequently, `y` is set to 6. Finally, `y=x*x++` sets `y` to 4. To give another example, `x++==x` always yields *false* as logical value.

Example: The following program counts the different vowels in a file and displays the result as list of type/token ratios. We shall refer to it as `countVowels` in the sequel.

```
#!/bin/sh
sed 's/[^aeiou]//g;    s/./&\
/g'  $1 |\
awk 'BEGIN { n["a"]=n["e"]=n["i"]=n["o"]=n["u"]=0;  OFS="\n" }
```

```
    { n[$1]++ }
END { print "a " n["a"],"e " n["e"],"i " n["i"],"o " n["o"],"u " n["u"] }'
```

The first `sed` program removes all non-vowels from the source file `$1` and isolates vowels by putting them on separate lines. We use an associative array `n` to count the number of occurrences of every vowel. We set the array elements to 0 in the first line of the program such that 0 and not the empty string is printed in case no corresponding vowel occurred. In the second line, e.g., `n["a"]` is increased by one if the line contains the string `a`. The final `print` statement prints the string `a`*blank*, then the content of the variable `n["a"]`, then an output field separator (i.e., a newline character), then the string `e`*blank*, then the content of the variable `n["e"]`. Etc.

Note: `countVowels` shows, in principle, how one can obtain statistics over occurrence of patterns using `awk`. Patterns such as phrases that contain a variable (e.g., `/In .*,? foreigners are many/`, to search for a common mistake of Japanese students of English [Webb 1992, p. 82]) can be identified with `sed` and/or `awk` and accounting is done in the way outlined in the preceeding example. Note however, that a probabilistic profile (frequency analysis) of a text in regard to letters is characteristic for languages. This can be used to decypher elementary cryptosystems (cf. [Koblitz 1994, p. 57]). We shall give a simpler version of the `awk` program in this example in the section about looping over associative arrays with a `for` statement (see the program `countFrequencies` below).

The precedence of multiplication over addition, polynomial expressions over inequalities, and inequalities (which can be understood as logical statements) over Boolean operations, give computational-algebraic expressions the format which is standard in, e.g., computer science or mathematics. After a computation is finished, a result can be assigned to a new variable through the assignment operator `=`. This explains the low precedence of the latter. When in doubt, one can use parenthesis to enforce any desired precedence.

Example: The following `sh` program computes the negative value of the frequencies of "items" (e.g., words or phrases) in a list of type/token ratios. This is useful in computing "distances" of a list of type/token ratios, if the latter are seen as vectors. We shall refer to it as `computeNegativeValue` in the sequel.

```
#!/bin/sh
awk  '{  $(NF)=-$(NF) ;  print  }'  $1
```

The sign of the last field is reversed. After that the pattern space is printed.

The operations listed above allow numerous other applications. Examples are all types of accounting where the lines from which numbers are extracted are picked by patterns. A file containing, among other things, lines of the form

```
GRADES:: student1  grade1  grade2  ...
```

can be used together with `$1`, `$2`, ... `$(NF)` to produce final grades and statistics over homework, exams etc. In a similar fashion, one can use (multi-line) records to maintain

any sort of accounts in a spreadsheet-like fashion. Such an approach saves the cost of buying software, allows the user to define a convenient format and allows the user to define any sort of operation. Finally, the format and the tables can be checked and maintained by machine automatically using `awk` together with `cron`. (Consult `man cron` and `man crontab`.) This includes automatic notification of due dates for certain entries in files and automatic generation of textual and graphic display of data. The latter can be achieved by generating properly formatted source files for, e.g., `groff`, `latex` or `mathematica`.

5.14 Functions

`awk` has the following built-in functions:

- `int sqrt exp log`

`int(`*expression*`)` is the integer part of *expression*. `sqrt()` is the square root function. `exp()` is the exponential function to base e and `log()` is its inverse. Consult [Lang 1983] for more details on these mathematical functions.
Example: In a positive or negative floating point number, everything after the period is truncated by `int`. Consequently, `int(-4.2)` yields `-4`.

- `length(`*string*`)`

returns the length of *string*, i.e., the number of characters in *string*. `length` is `length($0)`.
Example: `awk 'length>20'` prints all lines that contain more than 20 characters.

- `index(`*bigstring*`,`*substring*`)`

This produces the position where *substring* starts in *bigstring*. If *substring* is not contained in *bigstring*, then the value 0 is returned. This allows analysis of fields beyond matching a substring.

- `substr(`*string*$,n_1,n_2$`)`

This produces the n_1^{th} to the n_2^{th} character of *string*. If $n_2 >$`length(`*string*`)` or if n_2 is omitted, then *string* is copied from the n_1^{th} character to the end. This allows one to cut pieces out of strings.
Example: `awk '/ since / { print substr($0,index($0," since ")) }'` prints from lines containing the word "since" the tail of the line starting in that word.

- `split(`*string*`,name,"c")`

This splits *string* at every instance of the separator character `c` into the array `name` and returns the number of fields encountered.

- `string = sprintf(`*format* , *expr1* , *expr2 ...*`)`

This sets `string` to what is produced by `printf` *format* , *expr1* , *expr2 ...*

- `split(`*string*`,name,"c")`

Example: The following `sh` program determines the sum, the average and the standard deviation of the frequencies of items in a list `$1` of type/token ratios.

```
#!/bin/sh
sed  '/^[  ]*$/d'  $1  |\
```

```
awk  '{  s1+=$(NF) ;    s2+=$(NF)*$(NF)  }
END   {  print s1 ,  s1/NR ,  sqrt(s2*NR-s1*s1)/NR  }'
```

The `sed` program removes white lines in the source file which may occur. `s1` and `s2` are initiated automatically to value 0 by `awk`. `s1+=$(NF)` adds the last field in every line to `s1`. `s2+=$(NF)*$(NF)` adds the square of the last field in every line to `s2`. Thus, at the end of the program we have `s1`=$\sum_{n=1}^{\texttt{NR}}$`$(NF)`$_n$ and `s2`=$\sum_{n=1}^{\texttt{NR}}$(`$(NF)`$_n)^2$. In the `END`-line, the sum `s1`, the average `s1/NR` and the standard deviation (cf. [Gänssler & Stute 1977, p. 81]) are printed.

Example: Suppose that a source file `$1` contains whole sentences per line. Then the following `sh` program sorts non-white lines in `$1` into files in correspondence to logarithm to base 2 of the length of the sentences (i.e., number of fields). This yields a crude classification of sentences. If `$1`=`fName`, then, e.g., "We are all equal." is put into `fName.logLength2`.

```
#!/bin/sh
awk '/[^  ]/{L=int(log(NF+0.1)/log(2));F="'$1'.logLength" L;print >F}' $1
```

`L` is computed as the integer part of the logarithm to base 2 of `NF+0.1` in accordance with the formula $log_2(x) = log_e(x)/log_e(2)$. The term `+0.1` is used here to avoid round-down errors, i.e., $log_2(8)$ may be computed to be $int(2.999)$=2 by `int(log(NF)/log(2))` if `NF`=8. `F` is the filename which is generated through string concatenation.

Example: Suppose that lines in a file contain two fields separated by the character `#`. The first field `$1` contains German phrases and the second field `$2` contains the corresponding English translation. One may be interested in comparing the length (in words) of the German originals vs. the length of the English translation. The construct
`GermNumb=split($1,GermPhrase," ")` would count the number of fields (words) in the German original. In addition, the array `GermPhrase` would be created by `awk`. `GermPhrase[1]` would contain the first word in the German phrase, and `GermPhrase[GermNumb]` would contain the last word. `EnglNumb=split($2,EnglPhrase," ")` would allow the comparison.

5.15 Patterns II (patterns involving algebraic-logical expressions and functions)

Address patterns in `awk` that select the pattern space can be
1) regular expressions as described above similar to regular expressions in `sed`,
2) algebraic-computational expressions involving variables and functions, and
3) Boolean combinations of anything listed under 1) or 2).
Essentially, everything can be combined in a sensible way to customize a pattern.

Examples: `awk 'NR<43'` prints the first 42 lines in a file. This shows the typical way how to handle line numbers in address patterns.

`awk '$1+$2!=$3+$4'` prints all lines where the sum of the first two fields does not equal

the sum of the third and fourth field. (Recall that a string which does not represent a number has numerical value 0.)
`awk '$1>$2'` prints all lines where the first field is greater than the second field. If `$1` and `$2` cannot be clearly recognized as numbers, then they are compared as strings.

Such short appropriately designed `awk` programs can be used effectively to check files for inconsistencies. For example, whether or not due entries in a file have been made. Or whether or not a budget documented in a file is balanced. Such checks can be invoked automatically at specific instances in time using `cron`.

Example: `awk '$0~/^[^A-Za-z]*I[^A-Za-z.]/ && NF>15'` prints all lines starting with the word "I" which contain more than 15 fields.
`identifyBeginI fileName | awk 'NF>15'` does the same.

Example: The following `sh` program is another version of `adjustBlankTabs`.

```
#!/bin/sh
awk  'NF>0 { $1=$1; print } ;    NF==0 { print "" }'  $1
```

The statement `$1=$1` manipulates the pattern space containing `$0` for non-white lines. Then, `$0` is printed through `print` field-wise using `OFS`=*blank* by default. Thus, all fields are separated by exactly one blank. In case of a white line, an empty line is printed.

Example: The following `sh` program switches lines in a file. Lines 1 and 2 are exchanged, then lines 3 and 4 are exchanged, etc. We shall refer to it as `switchLines` in the sequel.

```
#!/bin/sh
awk  'NR%2==1 { oddLine=$0 } ;    NR%2==0 { print ; print oddLine }'  $1
```

If the line number is odd, then the line is stored in the variable `oddLine`. Otherwise, the current line is printed before the previous line is. Note that for a file with an odd number of lines, the last line is lost.

Example: The following `sh` program computes the absolute value of the frequencies of items (e.g., words or phrases) in a list of type/token ratios. This can be used to compute ℓ^1-distances of such lists using `computeNegativeValue` introduced above and `vectorAddition` listed below. It can be used to generate a distance map of students in a class in regard to vocabulary usage, if the usage of words by every student is documented in a list of type/token ratios. We shall refer to it as `computeAbsoluteValue` in the sequel.

```
#!/bin/sh
awk  '$(NF)<0 { $(NF)=-$(NF) } ;     { print }'  $1
```

If the last field is negative, then its sign is reversed. Next, every line is printed.

Example: The following `sh` program cuts away low frequencies that are below a limit value `$2` which is argument to `sh` if a file `$1` is a list of type/token ratios. We shall refer to it as `filterHighFrequencies` in the sequel. It can be used to gain files with very common words that are functional in the grammatical sense but not in regard to the context.

```
#!/bin/sh
awk  '$(NF)>='$2'+0'  $1
```

`$2` stands for the second argument to `sh`. If this program is invoked with `filterHighFrequencies fname 5`, then `sh` passes `$(NF)>=5+0` as selecting address pattern to `awk`. Consequently, all lines of `fname` where the last field is larger than or equal to 5 are printed.

Example: The next example is introduced to show the use of the functions `index()` and `substr()` in `awk`. It can be used to generate all possible sequences of consecutive words of a certain length in a file. In the sequel, we shall refer to it as `context`. Suppose that a file `$1` is organized in such a way that single words are on individual lines (e.g., the output of a pipe `leaveOnlyWords | oneItemPerLine`). `context` uses two arguments. The first argument `$1` is supposed to be the name of the file that is organized as described above. The second argument `$2` is supposed to be a positive integer. `context` then generates "context" of length `$2` out of `$1`. In fact, all possible sequences of length `$2` consisting of words in `$1` are concatenated and printed. Note that context can be seen either directed or as symmetric in the sense that the first word in one of the concatenated strings is in context to other words in the string.

```
#!/bin/sh
awk  'BEGIN { range='$2'+0 }
NR==1    { c=$0 };                          NR>1      { c=c " " $0 }
NR>range { c=substr(c,index(c," ")+1) };  NR>=range { print c    }'   $1
```

Suppose `$2`=11. In the first line of the `awk` program, the variable `range` is then set to 11. In the second statement of the `awk` program, the context `c` is set to the first word. In the third statement, a new word other than the first is appended to `c` separated by a blank. The fourth statement works as follows: after 12 words are collected in `c`, the first is cut away by using the position of the first blank, i.e., `index(c," ")` and reproducing `c` from `index(c," ")+1` until the end. Finally, the context `c` is printed, if it contains 11 words.

Note that the output of `context` has, essentially, eleven times the size of the input for the example just listed. It may be advisable to incorporate any desired, subsequent pattern matching for the strings that are printed by `context` into an extended version of this program.

Example: The following program gives an idea how to implement an inverse Polish notation calculator. It shows the combined use of address patterns and numerical computations.

```
#!/bin/sh
awk  'BEGIN      { x1=x2=x3=x4=0 }
      /^[0-9]+$/ { x4=x3 ; x3=x2 ; x2=x1 ; x1=$0 };
      /^\+$/     { x1+=x2 ; x2=x3 ; x3=x4  }
      /^e$/      { x1=exp(x1) };
                 { print x1 , x2 , x3 , x4 }'
```

First, the four register stack (x1,x2,x3,x4) with bottom x1 is set to 0 in order to have a nice print of the stack all the time. If a positive integer is entered (checked by /^[0-9]+$/), then a shift upwards is made in the stack storing the integer in x1 and losing x4. Observe the necessary inverse order of the assignments. If + (checked by /^\+$/) is entered, then x1 and x2 are added and stored in x1. The stack is shifted downwards retaining x4. If the letter e (checked by /^e$/) is entered, then the bottom of the stack is exponentiated. Finally, the stack is printed after every input. It is a good exercise to extend the above to include the full range of arithmetic operations, more types of signed integer and real numbers including 12.34E5 meaning $12.34 \cdot 10^5$, more functions, e.g, log_2 and trigonometric functions, arbitrary long, dynamic stack using an array, programming capability using another array, and commands to change from computing mode to programming mode. Also, consult man dc.

5.16 Matching fields of the input record

Very important is that fields of the input record can be matched using ~ (matches) and !~ (matches not). If fields are matched, then ^ and $ stand for the begin and end of the field.

Example: The following program selects lines whose last field is UK, U.K, UK. or U.K. and changes it to United Kingdom before printing the line.

```
#!/bin/sh
awk  '$(NF)~/^U\.?K\.?$/ {  $(NF)="United Kingdom" ;   print  }'  $1
```

Observe that $(NF)~/^UK$/ is as good as $(NF)=="UK" .

Note the equivalence of the following address patterns and conditionals. /MUSE/ is the same as $0~/MUSE/. !/MUSE/ is the same as $0!~/MUSE/. This sort of suppression of the ~ operator for $0 holds in general for address patterns but not in regard to conditionals in the if, for, resp. while statement. In connections with these, one has to use ~.

5.17 Address ranges

As in sed, one can specify a first address pattern where actions begin and a final pattern where actions end. These can be complicated conditionals involving all of the components mentioned in the last section. However, address patterns are usually quite simple.

Example: Suppose that code is inserted in a document starting with a line containing up to white characters only \BC and ending with a line containing up to white characters only \EC. Then, the following program prints all code and the framing \BC and \EC lines.

```
#!/bin/sh
awk  '/^[  ]*\\BC[  ]*$/,/^[  ]*\\EC[  ]*$/'  $1
```

The program uses the address range

```
/^[  ]*\\BC[  ]*$/,/^[  ]*\\EC[  ]*$/
```

starting with the address pattern `/^[ ]*\\BC[ ]*$/` and ending with
`/^[ ]*\\EC[ ]*$/`.

5.18 Elementary control structures (next exit)

`awk` has the usual control structures: `if`, `for` and `while`. In addition to these, `next` and `exit` are useful.

- `next`

is a statement that starts processing the next input line immediately from the top of the `awk` program. `next` is the analogue of the `b` command without address in `sed -n` or the `d` command in `sed`.

Example: `/`*address*`/ { next }` causes the termination of the current cycle, if the pattern space matches *address*. Such a thing is very useful, if one wants to exclude certain lines from being processed by subsequent actions in the cycle.

Example: The following program is another version of `switchLines`.

```
#!/bin/sh
awk 'NR%2==1 { oddLine=$0 ; next };    { print ; print oddLine }' $1
```

If the line number is odd, then the line is stored in the variable `oddLine` in the first statement. After that, the cycle is left through `next`, and the second statement of the `awk` program is deliberately missed. If the line number is even, then the address pattern in the first statement of the `awk` program is evaluated to *false*. Consequently, the second statement is reached and the current line is printed before the previous line is.

- `exit`

is a statement that causes `awk` to quit immediately. `exit` is the analogue of the `q` command in `sed`.

Example: `/`*address*`/ { quit }` causes the termination of the `awk` program, if the pattern space matches *address*.

5.19 The if statement

The `if` statement looks the same as in `C` [Kernighan & Ritchie 1988, p. 55]:

`if (`*conditional*`) {` *action1* `}`
`else {` *action2* `}`

conditional can be any of the types of conditionals we defined above for address patterns involving Boolean combinations of comparison of algebraic expressions including the use of variables. Note, however, that regular expressions `/`*regExpr*`/` that are intended to match resp. not intended to match the whole pattern space `$0` have to be used together with the matching operator `~`. Thus, one has to use `$0~/`*regExpr*`/` resp. `$0!~/`*regExpr*`/`. The `else`

part can be omitted or can follow on the same line. *action1* and *action2* can be spread over several lines. The corresponding closing braces `}` can follow on additional lines.

Example: To illustrate the use of `if`, we list another two versions of `switchLines`.

```
#!/bin/sh
awk  '{  if (NR%2==1) { oddLine=$0 }  else { print ; print oddLine }  }' $1
```

This version is identical to the first two up to notation. The next version illustrates the use of a flag variable (`f`).

```
#!/bin/sh
awk '{ if   (f==0) { oddLine=$0 ;             f=1 }
       else        { print ; print oddLine ; f=0 } }' $1
```

When the first line of the input is processed, then the variable `f` is initiated to 0. Thus, the conditional `f==0` yields *true* and the first part of the `if` statement is executed. In it, the line is stored in the variable `oddLine`, and `f` is set to 1 making sure that the conditional `f==0` fails for the next line (with even line number). If the conditional fails, then the `else` part of the `if` statement is executed. The current line (with even line number) is printed before the previous line is, and `f` is set to 0 making sure that `f==0` matches while processing the next line.

5.20 The for statement I (standard form of loops)

The `for` statement also looks the same as in `C` [Kernighan & Ritchie 1988, p. 60]:

`for (` *start* `;` *conditional* `;` *expression* `) {` *statements* `}`

Note the semicolons `;`. First, the expression *start* is executed. Then, *conditional* is checked. *conditional* is exactly as *conditional* in the `if` statement. If found to be *true*, then *statements* and after that *expression* are executed. Then, the cycle *conditional-statements-expression* is repeated until (hopefully) *conditional* fails. *statements* and the closing brace `}` can be spread over several lines. The individual statements in *statements* are separated by newlines or semicolons.

The most typical loop is looping over the fields in the pattern space as follows:

`for (f=1;f<=NF;f++) {` *actions with* `$(f) };`

First, the variable `f` is set to 1. Then, the *actions* are executed with the field variable `$(1)`. After that `f` is incremented to 2 by `f++`. If there was more than one field, then the *actions* are executed with `$(2)`. This cycle (increment of `f`, *actions with* `$(f)`) is continued until `f` is set to `NF`+1 scanning all fields from left to right.

Note that `$(f-1)` is the field preceeding `$(f)`, if `f`>2. Similarly, `$(f+1)` is the field following `$(f)`, if `f`<`NF`. In contrast to the above, `$f-1` is the *value* of `$f` minus 1.

One can of course loop backwards from right to left over all fields using
`for (f=NF;f>=1;f--) { `*actions with* `$(f) };`. Or one can loop over every second field using
`for (f=1;f<=NF;f+=2) { `*actions with* `$(f) };`.

Example: The following example is another version of `addBlanks`.

```
#!/bin/sh
awk  'BEGIN { ORS="" }
NF>0 {for(f=1;f<=NF;f++){ print " " $(f) " " }};  { print "\n" }'   $1
```

In a non-white line, every field is printed with framing blanks in the first `awk` statement in the last line. Since `ORS` is set to the empty string, the `print` statements do not generate new lines. As a price for that, a terminating newline character must be printed separately after the `for` loop. The latter is done for every line of input in the last `awk` statement. Consequently, only a newline character is printed for white lines.

Example: The following program is another version of `findFourLetterWords`. It shows a typical use of `for` and `if`, i.e., looping over all fields with `for` and on condition determined by `if` taking some action on the fields.

```
#!/bin/sh
leaveOnlyWords $1  |\
awk '{for(f=1;f<=NF;f++){ if($(f)!~/^[A-Za-z][a-z][a-z][a-z]$/){$(f)=""} }}
/[^ ]/ { print NR , $0 }'  |  adjustBlankTabs -
```

If a field `$(f)` does not match the pattern `/^[A-Za-z][a-z][a-z][a-z]$/`, then it is set to the empty string in the `for` loop. In case the pattern space stays non-white after this procedure, it is printed with a leading line number. Finally, blanks are properly adjusted by `adjustBlankTabs`.

Example: The following program is a better version of `context`. This version allows the concatenation of any sort of lines, not only lines that contain single words.

```
#!/bin/sh
awk  'BEGIN { r='$2'+0;  ORS="" }
            { line[NR%r]=$0     }
      NR>=r { for(c=1;c<=r;c++){print line[(NR+c)%r] " "}; print "\n"}' $1
```

In the first line of the `awk` program, the range `r` is set to the integer `$2` delivered by `sh`. The latest input line is always stored in the array `line`. The index for array changes cyclically modulo `r`. If `NR`$>$`r`, then the elements in `line` are cyclically overwritten. Thus, there are at most `r` elements in `line`. If `NR`$\geq$`r`, then the array `line` is printed by the `for` loop in proper order. The latest element read is printed at the end since `(NR+r)%r=NR%r`. Since `ORS=""`, separating blanks have to be listed explicitly, and a trailing newline character has to be printed separately after the `for` loop.

5.21 The for statement II (looping over associative arrays)

Let `array[`*index*`]` be an associative array where at least one string *index* has been used and thus `array[`*index*`]` has been set at least once in the `awk` program. The second form of the `for` statement is:

`for ( ind in array ) { `*statements*` }`

Again, *statements* `}` can be spread over several lines. If this form is used, then the (hopefully unused) variable `ind` is created and loops over the strings that were used as *index* with `array` in an undetermined order (randomly from the observers point of view). However, all strings for *index* that were used will be assumed by the variable `ind` exactly once. For every value of `ind`, *statements* is executed and `ind` is available with that particular value in that period of time.

Example: The next program is very useful for generating statistics. It is a simpler form of a program in [Kernighan & Pike 1984, pp. 123-124]. If the input is formatted, e.g., in such a way that every line contains exactly one word, then the output is a list of word frequencies encountered. The list of type/token ratios which is produced is arranged alphabetically. This short program yields the same results as, e.g., the rather long SNOBOL program given in [Butler 1985, pp. 206-208]. We shall refer to it as `countFrequencies` in the sequel.

```
#!/bin/sh
awk  'NF>0 { number[$0]++ }
END { for(string in number){ print string , number[string] }}' $1 | sort
```

The first line uses the string `$0` as an index. `$0` may contain blanks or tabs. Every time the string `$0` occurs, the counter `number[$0]` is increased by 1. This creates the associative array `number` whose indices are the strings `$0` on the lines which are counted by the value `number[$0]`. The `for` loop at the end of the program prints the statistics (i.e., number of occurrences) of the strings (items) in the type/token format which is then sorted by `sort`. The UNIX command `sort` puts the lines in the resulting list of type/token ratios in lexicographical order for words and phrases (which are presumably to the left of the final number in the line). Consult `man sort`. There exist many options for `sort`. One can, e.g., sort lines depending upon the numerical value of the second field. The next two examples show the potential of `countFrequencies`.

Example: The next program implements another, shorter version of `countVowels`. As in the previous version, vowels are put on separate lines by the first `sed` program.

```
#!/bin/sh
sed  's/[^aeiou]//g;    s/[aeiou]/&\
/g' $1  |  countFrequencies -
```

Example: The next program implements a word frequency count in a file `$1`. We shall refer to it as `countWordFrequencies` in the sequel.

```
#!/bin/sh
leaveOnlyWords $1 | oneItemPerLine - | mapToLowerCase - | countFrequencies -
```

Application: Let us show how `countWordFrequencies` can be used to implement a program that measures lexical density. [Butler 1985] provides lists of grammatical and ambiguous words (pp. 219-220) and a 216-line SPITBOL program to calculate lexical density. Suppose a combined list of grammatical and ambiguous words is kept in a file named `unimportantWords` and the text to analyze is kept in a file `theTextFile`. In addition, suppose that `unimportantWords` contains only one string of non-white characters per line. Then one can proceed as follows:

```
countWordFrequencies theTextFile |\
awk 'FILENAME=="unimportantWords" { n[$1]=1; next };
     n[$1]==0'  unimportantWords  -
```

`countWordFrequencies theTextFile` performs a word frequency count on the whole of `theTextFile`. The result of this is feed as second argument into the final `awk` program. As long as this `awk` program reads its first argument `unimportantWords`, it only creates an associative array `n` indexed by the words in `unimportantWords` with constant value 1 for the elements of the array. If the final `awk` program reads the standard input `-` containing the result of the word frequency count, then only those lines containing the field (word) `$1` are printed where `n[$1]` is initiated to zero by the conditional `n[$1]==0` which is then found to be *true*. For words from `unimportantWords`, the array element `n[$1]` already has the value 1 and the conditional `n[$1]==0` evaluates to *false* which means that the lines containing them are not printed.

5.22 The while statement

The `while` statement also looks the same as in `C` [Kernighan & Ritchie 1988, p. 60]:

`while (` *conditional* `) {` *statements* `}`

Again, *statements* `}` can be spread over several lines. *statements* is executed as long as *conditional* stays true.

5.23 Implementing set and vector operations

In this section, we shall give a few additional examples to illustrate the use of the `awk` constructs introduced above and to illustrate the use of pipes.

Set and vector operations have been useful in [Schmitt & Christianson 1998]. We have been interested in measuring students' progress in the use of global and specialized vocabulary. The University of Aizu is a "Computer Science University." Thus, we were interested in

measuring the progress of students in regard to certain target lists of words [Orr et al. 1995] that Japanese students should use and understand in order to communicate in the university's working language in their field of study. An alternative is to measure progress in regard to a general word list, such as the General Service List [Nation 1990]. Our procedure used set intersection. Also, it is useful to have an operation set union (`sort -u`) to collect, e.g., the vocabulary that is used by all students. We have used set as well as vector operations to implement various selection schemes.

Note that by maintaining a small list of words on-line for use in comparison with students' writings the teacher is afforded maximum flexibility. So, for instance, the teacher can add or delete words from the list, depending on the students' needs.

Example: The next program implements vector addition. We shall refer to it as `vectorAddition` in the sequel. If `aFile` and `bFile` are lists of type/token ratios, then it is used as `cat aFile bFile | vectorAddition`. It can be used to measure the cumulative advance of students in regard to vocabulary use.

```
#!/bin/sh
adjustBlankTabs $1 |\
awk 'NF>0 {  n=$(NF); $(NF)=""; sum[$0]+=n  }
     END  {  for (string in sum) { print string sum[string] }  }'  |  sort
```

In the first line of the `awk` program the last field in the pattern space `$0` is first saved in the variable `n` before the last field is set to the empty string retaining a trailing blank (*). An array `sum` is generated which uses the altered string `$0` in the pattern space as index. Its components `sum[$0]` are used to add up all corresponding numbers `n`. Recall that `sum[$0]` is initiated to 0 automatically. Finally, the associative array with looping index `string` is printed together with the sums and sorted into standard lexicographical order. Note that there is no comma in the `print` statement in view of (*).

Example: The next program implements set intersection. We shall refer to it as `setIntersection` in the sequel. A set is, by design or definition, represented as a file where the lines represent the distinct elements of the set. If `aFile` and `bFile` are non-empty sets in the sense just defined, then it is used as `setIntersection aFile bFile`. Note that `adjustBlankTabs fName | sort -u` converts any file `fName` into a set. In fact, `sort -u` sorts a file and only prints occurring lines at most once.

```
#!/bin/sh
awk  'FILENAME=="'$1'" { n[$0]=1; next };    n[$0]==1'  $1  $2
```

Suppose this command is invoked as `setIntersection aFile bFile`. This means `$1=aFile` and `$2=bFile` in the above. As long as this `awk` program reads its first argument `aFile`, it only creates an associative array `n` indexed by the lines `$0` in `aFile` with constant value 1 for the elements of the array. If the `awk` program reads the second file `bFile`, then only those lines `$0` in `bFile` are printed where the corresponding `n[$0]` was initiated to 1

while reading `aFile`. For elements which occur only in `bFile`, `n[$0]` is initiated to 0 by the conditional which is then found to be *false*. Note that changing the final conditional `n[$0]==1` to `n[$0]==0` implements set complement. If such a procedure is named `setComplement`, then `setComplement aFile bFile` computes all elements from `bFile` that are not in `aFile`.

5.24 Implementing a vocabulary trainer

The next `sh` program shows how to implement a prototype of training software for vocabulary. We shall refer to it as `vocabularyTrainer` in the sequel.

```
#!/bin/sh
awk 'BEGIN                    { print "To start, please, type RETURN." }
     FILENAME=="'$1'"&&NF==2 { quest[++c]=$1 ; answer[c]=$2 ; next }
     $0==answer[c]            { c-- }
     c<1                      { exit }
         { print "Please, translate the following word: ",quest[c] }'  $1 -
```

This program is activated as

```
vocabularyTrainer lessonFile
```

in a terminal, where `lessonFile` is a file containing pairs of words separated by blanks. The first column represents the question. The second column represents the answer or translation. A typical entry is as follows:
`leader Leiter`
First, the string `To start, please, type RETURN.` is printed. That is what the user sees in the terminal. Since `$1=lessonFile` is read first, the second line of the program is active first. This is so to speak the learning phase of the program. The counter `c` indexes questions and answers which are learned (memorized) by the program from `lessonFile`. Note that through the `next` statement the program never advances further in the cycle for a line from `lessonFile`. After reading `lessonFile` is finished, the input comes from standard input which is the terminal by default. First, the user hopefully types $\hookleftarrow$ (`RETURN`). This empty line is not matched by `$0==answer[c]`. `c` is at the highest possible value first. Thus, the last word is asked to be translated first by the fifth line of the `awk` program. If the user provides a correct answer to a question, then the counter is decremented by 1. If `c=0`, then the program exits at the top of `lessonFile`. Otherwise, the next query is presented (with the preceeding pair in `lessonFile`).

Possible extensions of this program are to quiz a student on the vocabulary repeatedly and to keep track of the user's performance in order to asks difficult words more often. Altogether, this shows that with relative ease one can develop prototype software and a surrounding/supporting file system with the tools presented in this paper. In the development

of education software, it is first more important to have a carefully designed database rather than a fancy display.

6 Further applications

6.1 Punctuation check and other applications to teaching English as a foreign language

This section uses examples specific to teaching English as a foreign language to native speakers of Japanese. Of course, any language teacher could modify the examples outlined here to fit a given teaching need.

Some of the first mistakes a teacher of English to Japanese students meets are purely mechanical: spelling and punctuation, especially when the students' writing is done on computers with English keyboards (as is the case at The University of Aizu). Japanese university students generally have little experience typing (in Japanese or English), and mechanical mistakes are abundant. Spelling errors can be corrected with the UNIX `spell` program. We have included in [Schmitt & Christianson 1998] automatic return of an evaluation with `spell` of the homework submitted via electronic mail in order to force students to use `spell` prior to submitting. We have also included a list of Japanese words that are acceptable when used in English text. Consult `man spell` for more details about `spell`. In [Schmitt & Christianson 1998], we reformat the result of the spell check which is sent back to the student using `sed` and `awk`.

More difficult to correct and teach is English punctuation, the rules of which, regarding spacing in particular, are different from Japanese. In fact, written Japanese does not include spaces either between words or after (or before) punctuation marks. At first, this problem may seem trivial. However, hours of class time spent discussing punctuation and yet more hours of correcting persistent errors manually tend to wear on teachers. Persistent errors in English punctuation have even been observed by one of the authors in English printing done by the Japan Bureau of Engraving, the government agency that typesets and prints the entrance examinations for Japanese universities. Clearly, if English punctuation rules (i.e., spacing rules) are not taught explicitly, they will not be learned.

A teacher using an automatic punctuation-correcting program such as the one in [Schmitt & Christianson 1998] described below is able to correct nearly all of the students' punctuation problems, thus presenting the spacing rules in an inductive, interactive way. We remark that a punctuation-correcting program is one of several tools described in [McDonald et al. 1988].

As a database, we have defined an exclusion matrix of forbidden 2-sequences (i.e, ordered pairs) of characters. During the setup phase of the system used in [Schmitt & Christianson 1998], this matrix is translated into a `sed` program which scans the essays submitted by

students via electronic mail for mistakes. These mistakes are marked, and the marked essays are send back to the individual students automatically. The translation into a `sed` program during setup works in the same way as the generation of an elimination program shown above. The resulting marking program is very similar to `markDeterminers`. Suffice it to say that this automated, persistent approach to correcting punctuation has been an immediate and dramatic success [Schmitt & Christianson 1998].

Finally, let us remark that our procedure to identify mistakes in punctuation can also be used in analyses of punctuation patterns, frequency, and use, as in [Meyer 1994].

6.2 Isolating sentences

In [Schmitt & Christianson 1998], one of the tools reformats the essays submitted by students in such a way that whole sentences are on single lines. In the examples given above, we have already demonstrated that such a format is very useful in two ways:
Goal 1: To select example sentences which match certain patterns and are actually submitted by students. The teacher can then write any number of programs that search for strings identified as particularly problematical for a given group of students. Furthermore, once those strings have been identified, the sentences containing them can be saved in separate files according to the strings and printed as lists of individual sentences. Such lists can then be given to students in subsequent lessons dealing with the problem areas for the students to read and determine whether they are correct or incorrect, and if incorrect, how to fix them.
Goal 2: To analyze example sentences. One example is to measure the complexity of grammatical patterns used by students using components such as `markDeterminers`. This can be used to show the decrease or increase of certain patterns over time using special `sed` based search programs and, e.g., `countFrequencies` as well as `mathematica` for display.

Our procedure of identifying sentences achieves a high level of accuracy without relying on proper spacing as a cue for sentence division, as does another highly accurate divider [Kaye 1990].

The following shows part of the implementation of sentence identification in [Schmitt & Christianson 1998]:

```
#!/bin/sh
hideUnderscore $1 | hideAbbreviations | hideNumbers | adjustBlankTabs |\
```

The implementations of `hideUnderscore` and `hideAbbreviations` have been discussed above. Compare also the listing of `leaveOnlyWords` given above. `hideNumbers` replaces, e.g., the string `$1.000.000` by `$1_000_000`, thus, "hiding" the decimal points in numbers. The next `sed` program listed below defines the ends of sentences. This is the most important component of the pipe which we show for reference.

```
sed 's/\([^]})'\''".!?][]}).!?]*\)\([!?]\)\([]})]*\)\([^]})'\''".!?]\)/\1\2\3__\2__\
```

```
\4/g
     s/\([^]})'\''".!?][]}).!?]*\)\([!?]\)\([]})]*\)$/\1\2\3__\2__/
s/\([^]})'\''".!?][]})'\''".!?]*\.[]})'\''"]*\)\([^]})'\''".!?]\)/\1__.__\
\2/g
  s/[^]})'\''".!?][]})'\''".!?]*\.[]})'\''"]*$/&__.__/' |\
```

In the first two `sed` commands, the ending of the sentence for "?" and "!" are defined. The letter ending the sentence is represented by `\2`. An additional newline character is introduced and the true ending of a sentence marked with `__\2__`. The last two substitution rules for marking sentences that end in the period are different than those for "?" and "!". In the `awk` program that follows, we merge lines that are not marked as sentence endings by setting the output record separator to a blank. If a line ending is marked, then an extra newline character is printed.

```
awk 'BEGIN   { ORS=" " }
             { print }
/__[!?.]__$/ { print "\n" }' |\ ...
```

Next, we merge all lines which start, e.g., in a lower case word with its predecessor since this indicates that we have identified a sentence within a sentence. Finally, markers are removed and the "hidden" things are restored in the pipe. By convention, we deliberately accept that an abbreviation does not terminate a sentence. Overall, our procedure creates double sentences on lines in rare cases. Nevertheless, this program is sufficiently accurate for the objectives outlined above in (1) and (2). Note that it is easy to scan the output for lines possibly containing two sentences and subsequently inspect a "diagnostic" file.

Let us conclude this section with an application: The string "and so on" is extremely common in the writing of Japanese learners of English, and it is objected to by most teachers. From the examples listed above such as `identifyBeginI`, `printPredecessorBecause` and `sortByVowel`, it is clear how to connect the output of the sentence finder with a program that searches for `and so on` and prints all sentences containing it in a separate file.

In [Webb 1992], 121 very common mistakes made by Japanese students of English are documented. We point out to the reader that a full 75 of these can be located in student writing using the most simple of string-search programs, such as those introduced above.

6.3 Judging the readability of texts

Hoey [Hoey 1991, pp. 35-48, pp. 231-235] points out that the more cohesive a foreign language text, the easier it is for learners of the language to read. One method Hoey proposes for judging relative cohesion, and thus readability, is by merely counting the number of repeated content words in the text (repetition being one of the main cohesive elements of texts in many languages). Hoey concedes though, that doing this by hand, i.e., a "rough

and ready analysis" [Hoey 1991, p. 235] is tedious work, impractical for texts of more than 25 sentences.

An analysis like this is perfectly suited for the computer, however. In principle, any on-line text could be analyzed in terms of readability based on repetition. One can use `countWordFrequencies` or a similar program to determine word frequencies over an entire text or "locally." Entities to search through "locally" could be paragraphs or all blocks of, e.g., 20 lines of text. The latter procedure would define a flow-like concept that could be called "local context." Words that appear at least once with high local frequency are understood to be important. A possible extension of `countWordFrequencies` is to use `spell -x` to identify derived words such as `Japanese` from `Japan`. Such a procedure aids teachers in deciding which vocabulary items to focus on when assigning students to read the text, i.e., the most frequently occurring ones ordered by their appearance in the text.

Example: The next program implements a search for words that are locally repeated in a text. In fact, we determine the frequencies of words in a file `$1` that occur first and are repeated at least three times within all possible strings of 200 consecutive words. 200 is an upper bound for the analysis performed in [Hoey 1991, pp. 35-48].

```
#!/bin/sh
leaveOnlyWords $1 | oneItemPerLine - | context - 200 |\
quadrupleWords - | countFrequencies -
```

`leaveOnlyWords $1 | oneItemPerLine | context - 200` generates all possible strings of 200 consecutive words in the file `$1`. `quadrupleWords` picks those words which occur first and are repeated at least three times within lines. `countFrequencies` determines the word frequencies of the so determined words.

Note again that `context - 200` creates an intermediate file which essentially has 200 times the size of the input. If one wants to apply the above to larger file, then the subsequent search in `quadrupleWords` should be combined with `context - 200`.

We have applied the above procedure to the source file of this document. Aside from functions words such as *the* and a few names, the following were found with high frequency: *UNIX*, *address*, *awk*, *character*, *command*, *field*, *format, liberal*, *line*, *pattern*, *program*, *sed*, *space*, *string*, *students*, *sum*, and *words*.

6.4 Lexical-etymological analysis

In [Gordon 1996], the author determined the percentage of etymologically related words shared by Serbo-Croatian, Bulgarian, Ukranian, Russian, Czech, and Polish. The author looked at 1672 words from the above languages to determine what percentage of words each of the six languages shared with each of the other six languages. He did this sort of analysis by hand using a single source. This kind of analysis can help in determining the validity of

traditional language family groupings, e.g.:

- Is the west-Slavic grouping of Czech, Polish, and Slovak supported by their lexica?
- Do any of these have a significant number of non-related words in its lexicon?
- Is there any other language not in the traditional grouping worthy of inclusion based on the number of words it shares with those in the group?

Information of this kind could also be applied to language teaching/learning by making certain predictions about the "learnability" of languages with more or less similar lexica and developing language teaching materials targeted at learners from a given related language (e.g., Polish learners of Serbo-Croatian).

Disregarding a discussion about possible copyright violations, it is easy today to scan a text source using a European type alphabet into computer obtaining automatically a file format that can be evaluated with a machine and, finally, do such a lexical analysis of sorting/counting/intersecting with the means we have described above. The source can be a text of any length. The search can be for any given (more or less narrowly defined) string or number thereof. In principle, one could scan in (or find on-line) a dictionary from each language in question to use as the source-text. Then one could do the following:
1) Write rules using `sed` to "level" or standardize the orthography to make the text uniform.
2) Write rules using `sed` to account for historical sound and phonological changes. (Such rules are almost always systematic and predictable. For example: the German intervocalic "t" is changed in English to "th." Exceptional cases could be included in the programs explicitly. All of these rules already exist, thanks to the efforts of historical linguists over the last century (cf. [Fox, 1995]).

Finally, there has to be a definition of unique one-to-one relations of lexica for the languages under consideration. Of course, this has to be done separately for every pair of languages.

6.5 Corpus exploration and concordance

The following `sh` program shows how to generate context of words from a file `$1`. In this example, two words are related if there are not more that `$2`−2 other words in between them. `$2` is supposed to be a strictly positive integer and the second argument to the program. If one so desires, then one could eliminate "unimportant" words from the text first. This sort of elimination procedure by automatically generated programs has been illustrated above in `eliminateList`.

```
#!/bin/sh
leaveOnlyWords $1 | oneItemPerLine - | mapToLowerCase - | context - $2 |\
awk '{ for (f=2;f<=NF;f++) { print $1,$(f) } }' | countFrequencies -
```

If a file contained the strings (words) `aa`, `ab`, `ac`, ... `zz` and `$2`=6, then the first line of output (into the pipe) of the first line in the above program would be `aa ab ac ad ae af`. The

`awk` program in the second line would then print `aa ab`, `aa ac`, ... `aa af` on separate lines. The occurrence of such pairs is then counted by `countFrequencies`. This defines a matrix M_d of directed context between the words in a text. One can obtain a symmetric context matrix M_s using the formula

$$M_s = M_d + M_d^{transpose}$$

Words $word_1$ and $word_2$ are distant or unrelated, if the frequency of the pair ($word_1$,$word_2$) in either M_d or M_s is low. Such a distance map can be displayed graphically using the `ListDensityPlot` command in `mathematica`. Note that building concordances, in particular for the Bible, is a very old endeavor. A search engine for concordance in Bible verses can be found on the internet in [Woods 1995].

Applying the procedure listed above to the source file of this document and filtering out low frequencies using `filterHighFrequencies - 20` the following were found among a long list containing otherwise mostly noise: (address pattern), (awk print), (awk program), (awk sed), (echo echo), (example program), (hold space), (input line), (liberal liberal), (line number), (newline character), (pattern space), (print print), (program line), (range sed), (regular expressions), (sed program), (sh awk), (sh bin), (sh program), (sh sed), (string string), and (substitution command).

Using the simple program listed above or some suitable modification, any language researcher or teacher can conduct basic concordancing and text analysis without having to purchase sometimes expensive and often inflexible concordancing or corpus-exploration software packages. For example, an analysis of collocations occurring with "between" and "through" as in [Kennedy 1991], which was conducted with "The Oxford Concordance Program OCP2" [Hockey & Martin 1988]), could very easily have been done with the following program providing, of course, that the user has an on-line corpus through which to search.

```
#!/bin/sh
leaveOnlyWords $1 | oneItemPerLine - | mapToLowerCase - | context - 20 |\
awk '(($1~/^between$/)||($(NF)~/^between$/))&&($0~/ through /)'
```

The second line of the program produces all sequences of consecutive words in the file `$1` of length 20 words. The `awk` program in the last line prints the sequences that contain `between` as first or last field and `through`.

One may chose not to discard the punctuation marks and to retain upper case letters. In that case, one can use

```
#!/bin/sh
oneItemPerLine $1 | context - 20 | awk '/[^A-Za-z][Tt]hrough[^A-Za-z]/' |\
awk '($1~/^[^A-Za-z]*[Bb]etween[^A-Za-z]*$/)||($(NF)~/^[^A-Za-z]*[Bb]etween[^A-Za-z]*$/)'
```

We have applied both procedures listed above to the source file of this document but found nothing which is not immediately related to this example.

In [Renouf & Sinclair 1991], a corpus search for the strings "`a [A-Za-z'-]* of`," "`an [A-Za-z'-]* of`," "`for [A-Za-z'-]* of`," "`had [A-Za-z'-]* of`," "`many [A-Za-z'-]* of`," "`be [A-Za-z'-]* to`," and "`too [A-Za-z'-]* to`" was conducted. Modifying the above program as follows would allow the user to search for these strings and print the strings themselves and ten words to both the right and left of the patterns in separate files.

```
#!/bin/sh
leaveOnlyWords $1 | oneItemPerLine - | mapToLowerCase - | context - 23 |\
awk '($(11)~/^((an?)|(for)|(had)|(many))$/)&&($(13)=="of") {
                                          F="'$1'." $(11) ".of"; print>F }
     ($(11)~/^((be)|(too))$/)              &&($(13)=="to") {
                                          F="'$1'." $(11) ".to"; print>F }'
```

We have applied the above procedure to the source file of this document. Mostly used are "a list of," "a collection of," and "a string of" for the first type of search. There are no strings matching the pattern `many [A-Za-z'-]* of` in the source. Mostly used are "be used to," "be set to," and "be applied to" for the second type of search. There are no strings matching the pattern `too [A-Za-z'-]* to` in the source.

It has been noted in several corpus studies of English collocation ([Kjellmer 1989], [Smadja 1989], [Atkins 1992]) that searching for 5 words on either side of a given word will find 95% of collocational co-occurrence in a text. After a search has been done for all occurrences of word $word_1$ and the accompanying 5 words on either side in a large corpus, one can then search the resulting list of surrounding words for multiple occurrences of word $word_2$ to determine with what probability $word_1$ co-occurs with $word_2$. The formula in [Collier 1993, p. 291] can then be used to determine whether an observed frequency of co-occurrence in a given text is indeed significantly greater than the expected frequency.

In [Christianson 1997], the English double genitive construction, e.g., "a friend of mine" is compared in terms of function and meaning to the preposed genitive construction "my friend." In this situation, a simple search for strings containing `of (()|()|...)` (*dative possessive pronouns*) and `of .*'s` would locate all of the double genitive constructions (and possibly the occasional contraction, which could be discarded during the subsequent analysis). In addition, a search for *nominative possessive pronouns* and `of .*'s` together with the ten words that follow every occurrence of these two grammatical patterns would find all of the preposed genitives (again, with some contractions). Furthermore, a citation for each located string can be generated that includes document title, approximate page number and line number. We have already discussed the practical, pedagogical application for this sort of concordancing programs.

6.6 Producing on-line dictionaries for the internet

In the course of the investigations outlined in [Abramson et al. 1995], [Abramson et al. 1996a], and [Abramson et al. 1996b], one of the authors developed a family of programs that are able to transform the source file of [Nelson 1962], which was typed with a *what-you-see-is-what-you-get* editor into a `prolog` database. In fact, any machine-readable format can now be generated by either slightly altering the programs already developed or by using `prolog` from now on.

The source was available in two formats: 1) an RTF format file, and 2) a control sequence free text file generated from the first file. Both formats have advantages and disadvantages. As outlined above, the RTF format file distinguishes Japanese *on* and KUN pronunciation from ordinary English text using *italic* and SMALL CAP typesetting, respectively. On the other hand, the RTF format file contains many control sequences that make the text "dirty" in regard to machine evaluation. We have already outline above how unwanted control sequences in the RTF format file were eliminated, but valuable information in regard to the distinction of *on* pronunciation, KUN pronunciation and English was retained. The second control sequence free file contains the standard format of kanji which is better suited for processing in the UNIX environment we used. In addition, this format is somewhat more regular already, which is useful in regard to pattern matching that identifies the three different categories of entry in [Nelson 1962]: *radical*, *kanji* and *compound*. However, very valuable information is lost in regard to the distinction of *on* pronunciation, KUN pronunciation and English.

We remark that parsers for RTF format files that are selective in the way which was needed here are not in existence.

Our first objective was to merge both texts line-by-line and to extract from every pair of lines the relevant information. Merging was achieved through pattern matching, observing that not all but most lines correspond one-to-one in both sources. Kanji were identified through use of the `sed` command `l`. As outlined above in the chapter on `sed`, control sequences were eliminated from the RTF format file but the information some of them represent was retained.

After the source files were properly cleaned by `sed` and the different pieces from the two sources identified (tagged), `awk` was used to generate a format from which all sorts of applications are now possible. The source file of [Nelson 1962] is typed regularly enough that, using pattern, matching the three categories of entry *radical*, *kanji* and *compound* can be identified. In fact, a small grammar was defined for the structure of the source file of [Nelson 1962] and verified with `awk`. Simply by counting all units, an index for the dictionary is generated which is a previously non-existing feature. This is useful in finding compounds in a search over the database and was previously impossible. In addition, all relevant pieces of data in the generated format can be picked by `awk` as fields and framed with, e.g., `prolog` syntax. What is now in the process of being completed is to reformat [Nelson 1962] such

that it can be used as an on-line database on the internet with a large range of forward and backward referencing. It is also easy to generate, e.g., English→*kanji* or English→KUN dictionaries from this *kanji*→*on*/KUN→English dictionary using `sort` and rearrangement of fields. In addition, it is easy to reformat [Nelson 1962] into proper `jlatex` format. This could be used to re-typeset the dictionary and to keep an updated version of the `jlatex` file as reference for all other components described in this section.

7 Conclusion

In the previous exposition, we have given a short but detailed introduction to `sed` and `awk` and their applications. We have shown that developing sophisticated tools with `sed` and `awk` is easy even for the computer novice. In addition, we have demonstrated how to write customized filters with particularly short code that can be combined in the UNIX environment to powerful processing devices particularly useful in language research.

Applications are searches of words, phrases, and sentences that contain interesting or critical grammatical patterns in any machine readable text for research and teaching purposes. We have also shown how certain search or tagging programs can be generated automatically from simple word lists. Part of the search routines outlined above can be used to assist the instructor of English as a second language through automated management of homework submitted by students through electronic mail. This management includes partial evaluation, correction and answering of the homework by machine using programs written in `sed` and/or `awk`. In that regard, we have also shown how to implement a punctuation checker.

Another class of applications is the use of `sed` and `awk` in concordancing. A few lines of code can substitute for a whole commercial programming package. We have shown how to set up searches that were performed with larger third-party packages in a simple way. Our examples include concordancing for pairs of words, other more general patterns, and the judgement of readability of text. The result of such searches can be sorted and displayed by machine for subsequent human analysis. Another possibility is to combine the selection schemes with elementary statistical operations. We have shown that the latter can easily be implemented with `awk`.

A third class of application of `sed` and `awk` is lexical-etymological analysis. Using `sed` and `awk`, dictionaries of related languages can be compared and roots of words determined through rule-based and statistical analysis.

Various selection schemes can easily be formulated and implemented using set and vector operations on files. We have shown the implementation of set union, set complement, vector addition, and other such operations.

Finally, all the above shows that `sed` and `awk` are ideally suited for the development of prototype programs in certain areas of language analysis. One saves time in formatting the

text source into a suitable database for certain types of programming languages such as `prolog`. One saves time in compiling and otherwise handling `C`, which is required if one does analysis with `lex` and `yacc`. In particular, if the developed program runs only a few times this is quite efficient.

8 Disclaimer

The authors do not accept responsibility for any line of code or any programming method presented in this paper. There is absolutely no guarantee that these methods are reliable or even function in any sense. Responsibility for any use of the methods presented in this paper lies solely in the domain of the applier. The programs listed in this paper are in the public domain, if not protected by copyright from third party. The terms of the software license agreement of the free software foundation
`ftp://prep.ai.mit.edu/pub/gnu/COPYING-2.0` apply.

9 References

Abramson, H., Bhalla, S., Christianson, K. T., Goodwin, J. M., Goodwin, J. R. & Sarraille, J. (1995). Towards CD-ROM based Japanese ↔ English dictionaries: Justification and some implementation issues. *Proceedings of the Third Natural Language Processing Pacific-Rim Symposium, Seoul, Korea, (Dec. 4-6, 1995)*, 174-179.

Abramson, H., Bhalla, S., Christianson, K. T., Goodwin, J. M., Goodwin, J. R., Sarraille, J. & Schmitt, L. M. (1996a). Multimedia, multilingual hyperdictionaries: A Japanese ↔ English example. Paper presented at the Joint International Conference of the Association for Literary and Linguistic Computing and the Association for Computers and the Humanities, Bergen, Norway, (June 25-29, 1996). `http://www.hd.uib.no/allc-ach.abstract.html`

Abramson, H., Bhalla, S., Christianson, K. T., Goodwin, J. M., Goodwin, J. R., Sarraille, J. & Schmitt, L. M. (1996b). The Logic of Kanji lookup in a Japanese ↔ English hyperdictionary. Paper presented at the Joint International Conference of the Association for Literary and Linguistic Computing and the Association for Computers and the Humanities, Bergen, Norway (June 25-29, 1996). `http://www.hd.uib.no/allc-ach.abstract.html`

Aho, A. V., Kernighan, B. W. & Weinberger, P. J. (1978). awk – A Pattern Scanning and Processing Language (Second Edition). In: B. W. Kernighan & M. D. McIlroy (Eds.) *UNIX programmer's manual (7th edition)*. Murray Hill, NJ: Bell Labs. On-line `troff` file: `http://cm.bell-labs.com/7thEdMan/vol2/awk`

Aho, A. V., Kernighan, B. W. & Weinberger, P. J. (1988). *The AWK programming language*. Reading, MA: Addison-Wesley Publishing Company.

Aijmer, K. & Altenber, B. (Eds.) (1991). *English corpus linguistics*. New York: Longman Publishers.

Allen, J. R. (1995). The ghost in the machine: Generating error messages in computer assisted language learning programs. *CALICO Journal*, 13 (2,3), 87-103.

Atkins, B. T. S. (1992). Tools for computer-aided corpus lexicography. *Acta Linguistica Hungarica*, 41 (1-4), 5-71.

Butler, C. (1985). *Computers in linguistics*. Oxford: Basil Blackwell Inc.

Christianson, K. T. (1997). A text analysis of the English double genitive. *IRAL* 35(2), 99-113.

Clocksin, W. F. & Mellish, C. S. (1981). *Programming in Prolog*. Berlin: Springer-Verlag.

Collier, A. (1993). Issues of large-scale collocational analysis. In: J. Aarts, P. De Haan, and N. Oostdijk (Eds.), *English language corpora: Design, analysis and exploitation*, 289-298. Amsterdam: Editions Rodopi, B.V.

Dougherty, D. (1990). *sed & awk – UNIX power tools*. Sebastopol, CA: O'Reilly & Associates.

Fox, A. (1995). *Linguistic Reconstruction: An Introduction to Theory and Method*. Oxford: Oxford University Press.

P. Gänssler, P. G. & Stute, W. (1977). *Wahrscheinlichkeitstheorie*. Berlin: Springer-Verlag.

Goldfield, J. D. (1986). An Approach to Literary Computing in French. In: *Méthodes quantitatives et informatiques dans l'étude des textes*, 457-465. Geneva: Slatkin-Champion.

Gordon, M. (1996). What does a language's lexicon say about the company it keeps?: A slavic case study. Paper presented at the Annual Michigan Linguistics Society Meeting, Michigan State University, East Lansing, Michigan, October, 1996.

Greenbaum, S. (1991). The development of the International Corpus of English. In: K. Aijmer & B. Altenberg (Eds.) *English corpus linguistics*, 83-91. New York: Longman Publishers.

Greub, W. (1981). *Graduate Texts in Mathematics 23: Linear Algebra.* Berlin: Springer-Verlag.

Herold, H. (1994). *UNIX und seine Werkzeuge awk und sed.* Reading, MA: Addison-Wesley Publishing Company.

Hockey, S. & Martin, J. (1988). *The Oxford concordance program: User's manual (Version 2).* Oxford: Oxford University Computing Service.

Hoey, M. (1991). *Patterns of lexis in text.* Oxford: Oxford University Press.

Hume, A. G. & McIlroy, M. D. (1990). *UNIX programmer's manual (10th edition).* Murray Hill, NJ: Bell Laboratories.

Johnson, S. C. (1978). Yacc: Yet another compiler-compiler. In: B. W. Kernighan & M. D. McIlroy (Eds.) *UNIX programmer's manual (7th edition).* Murray Hill, NJ: Bell Labs.
On-line `troff` file: `http://cm.bell-labs.com/7thEdMan/vol2/yacc.bun`

Kaye, G. (1990). A corpus builder and real-time concordance browser for an IBM PC. In: J. Aarts and W. Meijs (Eds.), *Theory and practice in corpus linguistics*, 137-161. Amsterdam: Editions Rodopi, B.V.,

Kennedy, G. (1991). *between* and *through*: The company they keep and the functions they serve. In: K. Aijmer & B. Altenberg (Eds.) *English corpus linguistics*, 95-110. New York: Longman Publishers.

Kernighan, B. W. & McIlroy, M. D. (1978). *UNIX programmer's manual (7th edition).* Murray Hill, NJ: Bell Laboratories. See: [Hume & McIlroy 1990] for a newer edition.

Kernighan, B. W. & Pike, R. (1984). *The UNIX programming environment.* Englewood Cliffs, NJ: Prentice Hall Inc.,

Kernighan, B. W. & Ritchie, D. M. (1988). *The C programming language.* Englewood Cliffs , NJ: Prentice Hall Inc.

Kjellmer, G. (1989). Aspects of English collocation. In: W. Meijs (Ed.), *Corpus linguistics and beyond*, 133-140. Amsterdam: Editions Rodopi, B.V.

Koblitz, N. (1994). *A course in number theory and cryptography.* Graduate texts in mathematics 114. Berlin: Springer-Verlag.

Lamport, L. (1986). *Latex – A document preparation system.* Reading, MA: Addison-Wesley Publishing Company.

Lang, S. (1983). *Undergraduate Analysis* Berlin: Springer-Verlag.

Leech, G. & Fligelstone, S. (1992). Computers and corpus analysis. In: C. S. Butler (Ed.), *Computers and written texts*, Oxford: Basil Blackwell Inc.

Lunde, K. (1993). *Understanding Japanese information processing.* Sebastopol, CA: O'Reilly & Associates.

Lesk, M. E. & Schmidt, E. (1978). Lex – A lexical analyzer generator. In: B. W. Kernighan & M. D. McIlroy (Eds.) *UNIX programmer's manual (7th edition).* Murray Hill, NJ: Bell Labs.
On-line `troff` file: `http://cm.bell-labs.com/7thEdMan/vol2/lex`

McDonald, N. H., Frase, L. T., Gingrich, P. & Keenan, S. (1988). The Writer's Workbench: Computer aids for text analysis. *Educational Psychologist* 17, 172-179.

McMahon, A. M. S. (1983). *Understanding Language Change.* Cambridge, UK: Cambridge University Press.

Meyer, C. F. (1994). Studying usage in computer corpora. In: G. D. Little and M. Montgomery (Eds.), *Centennial usage studies*, 55-61. Jacksonville, IL: American Dialect Society.

Meyer, C. F. & Tenney, R. L. (1993). Tagger: An interactive tagging program. In: C. Souter and E. Atwell (Eds.), *Corpus-based computational linguistics.* Amsterdam: Editions Rodopi, B.V.

Nation, I. S. P. (1990). *Teaching and learning vocabulary.* Boston: Heinle & Heinle Publishers.

Nelson, A. N. (1962). *The original modern reader's Japanese-English character dictionary (Classic edition).* Rutland, VT: Charles E. Tuttle Company.

Ossanna, J. F. & Kernighan, B. W. (1978). Troff user's manual. In: B. W. Kernighan & M. D. McIlroy (Eds.) *UNIX programmer's manual (7th edition).* Murray Hill, NJ: Bell Labs.
On-line `postscript` file: `http://cm.bell-labs.com/plan9/doc/troff.ps`

Orr, T., Christianson, K. T., DeHart, J., Izzo, J., Lambacher, S., Moore, C. H., Moore, W., Murakawa, H. & Washburn, N. (1995). A list of words and phrases related to computer science. Unpublished manuscript. The Center for Language Research, The University of Aizu, Aizu-Wakamatsu, Japan.

Renouf, A. & Sinclair, J. M. (1991). Collocational frameworks in English. In: K. Aijmer & B. Altenberg (Eds.) *English corpus linguistics*, 128-143. New York: Longman Publishers.

Schmitt, L. M. & Christianson, K. T. (1998). Pedagogical Aspects of a UNIX-Based Network Management System for English Instruction. *System* 26(4), to appear.

Smadja, F. A. (1989). Lexical co-occurrence: The missing link. *Literary and Linguistic Computing*, 4 (3), 163-168.

Wall, L. & Schwarz, R. L. (1990). *Programming perl.* Sebastopol, CA: O'Reilly & Associates.

Webb, J. H. M. (1992). 121 *common mistakes of Japanese students of English (Revised edition).* Tokyo: The Japan Times.

Wolfram, S. (1991). *Mathematica – A system for doing mathematics by computer (2nd edition).* Reading, MA: Addison-Wesley Publishing Company.

Woods, S. (1995). *Web chapel Bible concordance.*
www-page at `http://web2.airmail.net/webchap/wcconc.html`.