12

Linguistic Computing with UNIX Tools

Lothar M. Schmitt, Kiel Christianson, and Renu Gupta

12.1 Introduction

This chapter presents an outline of applications to language analysis that open up through the combined use of two simple yet powerful programming languages with particularly short descriptions: `sed` and `awk`. We shall demonstrate how these two UNIX[1] tools can be used to implement small, useful and customized applications ranging from text-formatting and text-transforming to sophisticated linguistic computing. Thus, the user becomes independent of sometimes bulky software packages which may be difficult to customize for particular purposes.

To demonstrate the point, let us list two lines of code which rival an application of "The Oxford Concordance Program OCP2" [21]. Using OCP2, [28] conducted an analysis of collocations occurring with "between" and "through." The following simple UNIX pipe (*cf.* section 12.2.3) performs a similar analysis:

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

Each of the programs in the above pipe shall be explained in detail in this chapter and contains 3 to 12 essential[2] lines of code.

This chapter is a continuation of [40] where a short, more programming-oriented tutorial introduction to the use of `sed` and `awk` for language analysis can be found including a detailed listing of all operators. A large number of references to [40] including mirror-listings can be found on the internet. A recommended alternative to consulting [40] as supplement to this chapter is reading the manual pages[3] for `sed` and `awk`. In addition, it is recommended (but not necessary) to read [4] and the introductions to `sed` and `awk` in [30].

[1] The term UNIX shall stand in the remainder of this chapter for "UNIX or LINUX."

[2] The procedure `leaveOnlyWords` is lengthy because it contains one trivial line of code per single-period-abbreviation such as "`Am.`". It can be computer-generated using `sed` from a list of such abbreviations (*cf.* section 12.3.4).

[3] Type `man sed` and `man awk` in a UNIX terminal window under any shell.

In order to use sed and awk effectively for language analysis, only limited knowledge of UNIX is needed. For proof of this claim and the convenience of the reader, we have listed 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. See Section 12.2.

An underlying principle in many of the applications presented in this chapter is the fact that the pipe mechanism of UNIX allows the manipulated text to be not only data, but to interact with the UNIX commands/filters in the pipe. From this perspective, the pipe and data are "fluid" rather than static objects. This allows for a very efficient programming style.

Included in this work 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, sed and awk. 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 UNIX tools such as lex [34, 30] or yacc [25, 30], and programming languages such as C [31] or prolog [11] can be used together with the methods presented here. See also [10] for an introduction of certain UNIX tools for language analysis. Note that, *e.g.*, mathematica [47] or some public domain plotting software such as gnuplot [17] can be used to generate graphics from numerical data produced with awk.

The latter parts of this chapter describe methods of application. Some of these applications have been used in [39] 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 and automatic correction of short essays that were submitted as homework by Japanese students of English composition via e-mail. One can use sed and awk to isolate phrases and sentences that were submitted by students and contain critical or interesting grammatical patterns for presentation in class. Other applications of sed and awk in [39] are the analysis of grammatical patterns in students' writings as well as statistical evaluation.

In addition to the applications just described, we can show how to set up a language training environment (*cf.* [40]), how to develop tools for statistical evaluation of text, be it in regard to concordance (*cf.* [28] or the example listed above, [38]), in regard to lexical-etymological analysis (*cf.* [19]), or in regard to judging the readability of text (*cf.* [22]). In [38], 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 [19] by hand. And, we shall describe how our procedure which counts word frequencies can be used to roughly judge the readability of text (*cf.* [22]). 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: [37]) into a multi-dimensional database for the internet. In addition, our exposition contains many comments in regard to other applications 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. Scattered reference to sed and awk can be found in descriptions of literary computing, *e.g.*, [18], 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 with the exception of [39]. We shall demonstrate that `sed` and `awk` provide easy-to-understand means to use programming in linguistic research. A genuine alternative to the approach presented in this chapter is using `perl` [44].

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.

12.2 Implementing a Word Frequency Count Using the Bourne-Shell

One can activate the Bourne-shell `sh` by typing `sh`↩ in a terminal window on a computer running the UNIX/LINUX operating system. The Bourne-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` in `sh` causes `sh` to terminate. For `sh`, a list of 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`. This is done, in particular, for the purpose of demonstrating how little initial knowledge of UNIX is necessary to start using `sed` and `awk` programming in linguistic research.

12.2.1 Creating a UNIX Command Using the Bourne-Shell

Essentially, a `sh` program is a file containing one `sh` command per line as well as some 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 lines into a file `lowerCaseStrings` with your favorite text editor:

```
#!/bin/sh
# Comment:  lowerCaseStrings
# (1) Map upper-case characters to lower case.
# (2) Isolate strings of non-white characters on separate lines.
sed  'y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/
s/[^a-z][^a-z]*/\
/g' $1
```

Save this file as `yourLoginName/lowerCaseStrings` directly in your home directory. Then, type `cd ; chmod 700 lowerCaseStrings` in your currently used shell window[4]. `lowerCaseStrings` is now a UNIX command just like the built-in ones. Next, type `lowerCaseStrings yourTextFile` in your shell window to see what the

[4] `cd` sets your working directory to your home directory. `chmod 700 lowerCaseStrings` makes `lowerCaseStrings` executable for you in addi-

program does. Here, `yourTextFile` should be a smaller plain-text file in your home directory, and the output of the command will appear in your shell window. It can be redirected to a file (*cf.* Section 12.2.3).

Explanation: The first line `#!/bin/sh` of `lowerCaseStrings` tells whatever shell you are using that the command lines are designed for `sh` which executes the file. The next three lines are comment. Comment for `sh`, `sed` and `awk` starts by definition with a `#` as first character in a line. Note that comment is not allowed within a multi-line `sed` command. The last three lines of `lowerCaseStrings` are one `sh` command which calls `sed` and delivers two arguments (subsequent strings of characters) to `sed`. The first entity following `sed` is a string of characters limited/marked by single-quote characters `'` which constitutes the `sed` program. Within that program, the `sed` command `y/ABC...Z/abc...z/` maps characters in the string `ABC...Z` to corresponding characters in the string `abc...z`. In the remainder of this paragraph, the italicized string '*newline*' stands for the "invisible" character that causes a line-break when displayed. The `sed` command `s/[^a-z][^a-z]*/\`*newline*`/g` substitutes(`s`) every(`g`) string of non-letters by a single *newline*-character (encoded as[5] \*newline*). A string of non-letters (to be precise: non-lower case letters) is thereby encoded as *one non*(`^`)*-letter* `[^a-z]` followed by *an arbitrary number*(`*`) *of non-letters* `[^a-z]*`. Consequently, `s/[^a-z][^a-z]*/\`*newline*`/g` puts all strings of letters on separate lines. The trailing `$1` is the second argument to `sed` and stands for the input-file name.[6] In the above example, one has `$1=yourTextFile`.

Remark: We shall refer to a program similar to `lowerCaseStrings` that only contains the first line of the `sed` program invoking the `y` operator as `mapToLowerCase`.

12.2.2 Implementing a Frequency Count Using awk

The above `sed` program combined with `awk` makes it easy to implement a simple word frequency count. For that purpose, we need a counting program which we shall name `countFrequencies`. The listing of `countFrequencies` shows the typical use of an array (here: `number`) in `awk` (*cf.* Section 12.4.1.3).

```
#!/bin/sh
# countFrequencies (Counting strings of characters on lines.)
awk '{ number[$0]++ }
     END { for (string in number) { print string , number[string] }}
    '  $1
```

Explanation: `awk` operates on the input (file) line by line. The string/symbol `$0` stands for the content of the line that is currently under consideration/manipulation.

tion to being readable and writable. Consult the manual pages entries (*i.e.*, type `man cd` and `man chmod` in your shell window) for further details about `cd` and `chmod`.

[5] \newline is seen as *newline* character by `sed`. *newline* alone would be interpreted as the start of a new command line for `sed`.

[6] More precisely: the trailing `$1` is the symbol the Bourne-shell uses to communicate the *first* string (*i.e.*, argument) after `lowerCaseStrings` to `sed` (*e.g.*, `yourTextFile` in `lowerCaseStrings yourTextFile` becomes `sed '`*program*`' yourTextFile`). Arguments to a UNIX command are strings separated by white space. Nine arguments `$1...$9` can be used in a UNIX command.

The first `awk` command `{ number[$0]++ }` increments a counter variable `number[`*string*`]` by 1, if the *string* is the content of the current line(=`$0`). For every occurring *string*, the counter `number[`*string*`]` is automatically initiated to 0. The character sequence `++` means "increase by one." If every line contains a single word, then at the end of the file, the counter variable `number[`*word*`]` contains the number of occurrences of that particular *word*. The `awk` command in the last line prints the *string*s which were encountered together with the number of occurrences of these *string*s at the `END` of processing. As in Section 12.2.1, the trailing `$1` stands for the input file.

12.2.3 Using the Pipe Mechanism

Combining `lowerCaseStrings` and `countFrequencies`, we create a UNIX command `wordFrequencyCount` as follows:

```
#!/bin/sh
# wordFrequencyCount
lowerCaseStrings $1 >intermediateFile
countFrequencies intermediateFile
```

The command `wordFrequencyCount` is used as `wordFrequencyCount tFile` where `tFile` is any plain-text file.

Explanation: `lowerCaseStrings $1` applies `lowerCaseStrings` to the first argument (string, filename) after `wordFrequencyCount` (*cf.* Section 12.2.1). The resulting output is then written/redirected via `>` to the file `intermediate File`, which is created if non-existent and *overwritten* if in existence[7]. `inter mediateFile` stays in existence after `wordFrequencyCount` terminates and can be further used. Finally, `countFrequencies intermediateFile` applies the word count to the intermediate result.

Instead of using `intermediateFile`, one can let the UNIX system handle the transfer (piping) of intermediate results from one program to another. The following `sh` program is completely equivalent to the first listing of `wordFrequencyCount` except that the intermediate result is stored nowhere:

```
#!/bin/sh
# wordFrequencyCount (2nd implementation)
lowerCaseStrings $1 | countFrequencies -
```

Explanation: The pipe symbol `|` causes the transfer (piping) of intermediate results from `lowerCaseStrings` to `countFrequencies`. The pipe symbol `|` or the string `|\` can terminate a line, in which case the pipe is continued into the next line. The trailing hyphen symbolizes the virtual file (in UNIX-jargon called "standard input") that is the input file for `countFrequencies`.

We observe that the output of `countFrequencies` is not sorted. The reader may want to replace the last line in the program by

```
lowerCaseStrings $1 | countFrequencies - | sort -
```

employing the UNIX command `sort` as the final step in the processing.

Additional information about programming `sh` and the UNIX commands mentioned above can be obtained using the `man sh` command as well as consulting [30].

[7] `>>` instead of `>` appends to an existing file.

12.3 Linguistic Processing with `sed`

The stream **ed**itor `sed` is the ideal tool to make replacements in texts. This can be used to mark, isolate, rearrange and replace strings and string patterns in texts. In this section, we shall exploit `sed`'s capabilities to present a number of small useful processing devices for linguistic computing which range from preprocessing devices to grammatical analyzers. All of these applications are essentially based upon simple substitution rules.

In our philosophy of text processing, the text itself becomes, *e.g.*, through the introduction of certain markers a program that directs the actions of the UNIX programs that act on it in a pipe.

12.3.1 Overview of `sed` Programming

A `sed` program operates on a file line-by-line. Roughly speaking, every line of the input-file is stored in a buffer called 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*. Each `sed` operator 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` operators are always applied to the current content of the pattern space and not the original input line. After the cycle is over, the resulting pattern space is printed/delivered to output, *i.e.*, the output file or the next process in the UNIX pipe mechanism. Lines that were never worked on are consequently copied to output by `sed`.

Substitution Programs

The simplest and most commonly used `sed` programs are short substitution programs. The following example shows a program that replaces the patterns `thing` and `NEWLINE` matching the strings `thing` and `NEWLINE` in all instances in a file[8] by `NOUN` and a *newline* character, respectively:

```
#!/bin/sh
sed  's/thing/NOUN/g
s/NEWLINE/\
/g'  $1
```

Explanation: The setup of the entire `sed` program and the two substitution commands of this program are very similar to the example in section 12.2.1. The first `sed` command `s/thing/NOUN/g` consists of four parts: (1) `s` is the `sed` operator used and stands for "substitute." (2) `thing` is the pattern that is to be substituted. A detailed listing of legal patterns in `sed` substitution commands is given in Appendix A.1. (3) `NOUN` is the replacement for the pattern. (4) The `g` means "globally." Without the `g` at the end 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 in text. This can be used to break pieces of text into fragments on separate lines for further separate processing. There is nothing following the trailing backslash `\` which is part of the `sed` program. See the

[8] As before, the symbol/string `$1` stands for the filename.

program `lowerCaseStrings` of section 12.2.1 for a non-trivial example using this technique.

We observe that one can also store `sed` commands without the two framing single quotes in a file (say) `sedCommands` and use `sed -f sedCommands` instead of or in any `sh` program as above.

Applications (tagging programs for grammatical analysis): The above example indicates how to examine a text for the overall structure of grammatical patterns occurring in that text. Thereby, lists of verbs, nouns and other grammatical entities can be automatically transformed into `sed` programs that perform identification (see, *e.g.*, the listing of the UNIX command `eliminateList` in Section 12.3.4). Similarly, a dedicated list of words can be automatically formatted into a (search) program in another programming language.

Applications (synonyms/translations through substitution): A custom-de- signed `sed` program similar to the above can be used to replace every word in a file by a bracketed list of synonyms. This can be used as a feedback device to encourage students to use a diversified vocabulary. In addition, note that `sed` can handle Japanese kanji characters. Thus, a custom-designed `sed` program similar to the above where every English word is replaced by a bracketed family of romaji, hiragana/katakana and kanji characters can be used to assist translation of text documents.

Application (cleaning files of control sequences in preprocessing): The replacement in a substitution can be empty. This can, *e.g.*, be used to "clean" a `tex/latex` [33] file of control sequences (*e.g.*, `\subsection`).

The Format of an Addressed `sed` Command

The format of an addressed `sed` command is *Address*`Command` *Address* can be omitted. In that case, `Command` is then applied to every pattern space. (On first reading, one may think of the pattern space as being the current line of input). If an *Address* is given, then `Command` is applied to the pattern space only if the pattern space matches *Address. Address* can be a pattern (regular expression) enclosed within slashes `/` as described in Appendix A.1, a line number not enclosed within slashes or `$` standing for the last line. In addition, a range of addresses in the format *StartActionAddress*,*EndActionAddress*`Command` can be used. In that case, `Command` is applied to every pattern space after *StartActionAddress* has been found until and including the pattern space where *EndActionAddress* has been found.

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
```

`sed` commands are terminated by either an immediately following *newline* character, a semicolon, or the end of the program.

One may wish to process the single quote `'` using `sed` or `awk`. In a program similar to the one listed above, the single quote `'` needs to be encoded as: `'\''`. This means for `sh`: (1) terminate the string listing the `sed` program temporarily at the first `'`, (2) concatenate the latter with the literal (\) character `'` and (3) continue the string listing the `sed` program by concatenating with the string following the third `'`. If a `sed` or `awk` program is stored in a file, then the the single quote `'` is encoded as itself. The representation of the slash `/` and backslash characters in `sed` programs are `\/` and `\\` respectively (*cf.* Appendix A.1).

Application (conditional tagging): A `sed` program similar to the program above can be used for conditional tagging. For example, if a file contains one entire sentence per line, then an *Address* can be used to conditionally tag (or otherwise process) certain items/words/phrases in a sentence depending whether or not that sentence contains a certain (other) key-item that is identified by the *Address* in the `sed` command.

12.3.2 Preprocessing and Formatting Tools

The next simple examples show how text can be preprocessed with small, customized `sed` programs such that the output can be used with much more ease 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.

Application (adding blanks for easier pattern matching): The following `sh` program adjusts blanks and tabs in the input file (symbolized by `$1`) in such a way that it is better suited for certain searches. This program will be often used in what follows since it makes matching items considerably easier. 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
# addBlanks
sed  's/[  ][  ]*/  /g;    s/^ */ /;
      s/ *$/ /;            s/^ *$//'   $1
```

Explanation: First, all strings consisting only of blanks or tabs are normalized to 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 in the last substitution command.
Justification: Suppose one wants to search in a file for occurrences of the word "liberal." In order to accurately identify the strings `Liberal` and `liberal` in raw text, one needs the following four patterns (compare Appendix A.1):

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

If one preprocesses the source file with `addBlanks`, only the first pattern is needed. Thus, a `sed`-based search program for `Liberal` and `liberal` is shorter and faster.

Application (Finding words in a text in a crude fashion): 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, a slash (*e.g.*, A/C) or an apostrophe are not properly identified.

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

Application (Putting non-white strings on separate lines): 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. This is a very good input format for counting and statistical operations on words. All ranges in the following program `[  ]` contain a blank and a tab. We shall call this `oneItemPerLine`.

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

Explanation: First, all white lines are removed by deleting the pattern space (`sed` operator `d`) which includes terminating the cycle[9], *i.e.*, the remainder of the `sed` program is not applied to the current pattern space, the current pattern space is not printed to output, and processing continues with the next line of input. 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.

Remark: Let us note at this point, that `sed` also has an operator to terminate the program. This is the operator `q` (quit). For example, `sed '5q' fName` prints the first 5 lines of the file `fName`, since it quits copying lines to the output (no action) at line 5.

Application (Normalizing phrases/items on separate lines): 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
# adjustBlankTabs
sed  's/^[  ]*//;   s/[  ]*$//;   s/[  ][  ]*/ /g'  $1
```

Explanation: All leading and trailing white space (blanks and tabs) is removed first. Finally, all white strings are replaced by a single blank in the last substitution command.

Justification: `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 or derive statistics for phrases which should be processed as unique strings of characters.

Technique: The following program replaces `@` by `@@`, `#` by `#@`, and `_` by `##` in an input file, *i.e.*, each of the single characters `@`, `#`, and `_` is replaced by the corresponding pair (consisting of characters `@` and `#` only) in the order of the substitution commands from left to right. In what follows, we shall refer to this program as `hideUnderscore`.

```
#!/bin/sh
# hideUnderscore
sed  's/@/@@/g;   s/#/#@/g;   s/_/##/g'  $1
```

[9] See the definition of "cycle" at the beginning of Section 12.3.1.

The following program is the inverse of `hideUnderscore`. In what follows, we shall refer to this inverse program as `restoreUnderscore`. Observe for the verification of the program that `sed` scans the pattern space from left to right.

```
#!/bin/sh
# restoreUnderscore
sed  's/##/_/g;    s/#@/#/g;    s/@@/@/g'  $1
```

Application (using a hidden character as a marker in text): 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 to mark exceptions. See the use of this technique in the implementations of `leaveOnlyWords` and `markDeterminers` in Section 12.3.3. Entities that have been recognized in text can be marked by keywords of the sort `_NOUN_`. 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 or properly formatted, and the "missing" character is restored.

Another application is to recognize the ends 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 a marker for the ends of sentences is, with minor additions, one way to generate a file which contains one entire sentence per line.

12.3.3 Tagging Linguistic Items

The tagged regular expression mechanism is the most powerful programming device in `sed`. This mechanism is not available in such simplicity in `awk`. It can be used to extend, divide and rearrange patterns and their parts. Up to nine chunks of the pattern in a substitution command can be framed (tagged) using the strings `\(` and `\)`.

Example: Consider the pattern `/[0-9][0-9]*\.[0-9]*/` which matches decimal numbers such as `10.` or `3.1415`. Tagging the integer-part `[0-9][0-9]*` (*i.e.*, what is positioned left of the period character) in the above pattern yields `/\([0-9][0-9]*\)\.[0-9]*/`.

The tagged and matched (recognized) strings can be reused in the pattern *and* the replacement in the substitution command as `\1`, `\2`, `\3` ... counting from left to right. We point out to the reader that the order of `\1`...`\9` standing for tagged regular sub-expressions need not be retained. Thus, rearrangement of tagged expressions is possible in the replacement in a substitution command.

Example: The substitution command `s/\(.\)\1/DOUBLE\1/g` matches double characters such as `oo`, `11` or `&&` in the pattern `/\(.\)\1/` and replaces them with `DOUBLEo`, `DOUBLE1` or `DOUBLE&` respectively. More detail about the usage of tagged regular expressions is given in the following three examples.

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

```
 1: #!/bin/sh
 2: # leaveOnlyWords
 3: sed  's/[^A-Za-z.'\''/-][^A-Za-z.'\''/-]*/ /g
 4: s/\([A-Za-z][A-Za-z]*\)\.\([A-Za-z][A-Za-z]*\)\./\1_\2_/g
 5: s/\([A-Za-z][A-Za-z]*_[A-Za-z][A-Za-z]*\)\./\1_/g
 6: s/Am\./Am_/g;    s/Ave\./Ave_/g;    s/Bart\./Bart_/g;
 7: # The list of substitution commands continues  ...
 8: s/vols\./vols_/g;    s/vs\./vs_/g;    s/wt\./wt_/g;
 9:                                          s/\./ /g;   s/_/./g
10: s/\([A-Za-z]\)\-\([A-Za-z]\)/\1_\2/g;   s/\-/ /g;   s/_/-/g
11: s/\([A-Za-z]\)\/\([A-Za-z]\)/\1_\2/g;   s/\-/ /g;   s/_/\//g
12: s/\([A-Za-z]\)'\''\([A-Za-z]\)/\1_\2/g; s/'\''/ /g; s/_/'\''/g
13: '    $1
```

Explanation: First, all strings which do not contain a letter, a period, an apostrophe, a slash or a hyphen are replaced by a blank (line 3). At this moment, the pattern space does not contain any underscore character which is subsequently used as a marker. The marker (`_`) is first used to symbolize period characters that are a part of words (abbreviations) and need to be retained. Next (lines 4–5), 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 6-8) comes a collection of substitution commands that replaces the period in standard abbreviations with an underscore character. Then (line 9), all remaining period characters are replaced by blanks (deleted) and subsequently all underscore characters by periods (restored). Next (line 10), every hyphen which is embedded between two letters is replaced by an underscore character. All other hyphens are then replaced by blanks (deleted), and subsequently all underscore characters are replaced by hyphens (restored). Finally (lines 11–12), the slash (encoded as `\/`) and the apostrophe (encoded as `'\''`, *cf.* Section 12.3.1) are treated in a similar way as the hyphen.

Example: The following program finds all four-letter words in a text. The program shows the usefulness of, in particular, `addBlanks` in simplifying pattern matching. We shall refer to it as `findFourLetterWords`.

```
#!/bin/sh
# findFourLetterWords (sed version)
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/ /' -
```

Explanation: 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) Using the `sed`-operator `=`, 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 output is piped into the second `sed` program which merges corresponding numbers and lines: 1) Using the `sed`-operator `N` (new line appended), every second line in the pipe (*i.e.*, every line coming from the original source file `$1` which contains at least one four-letter word) is appended via `N` to the preceding line in the pipe containing solely the corresponding line number. 2) The embedded newline character (encoded as `\n`) is removed and the two united lines are printed as one.

Application (tagging grammatical entities): The following program shows the first serious linguistic application of the techniques introduced so far. It marks all determiners in an input text file symbolized by `$1`. We shall refer to it as `markDeterminers`.

```
 1: #!/bin/sh
 2: # markDeterminers
 3: addBlanks $1 | sed  ’s/\.\.\./_TRIPLE_PERIOD_/g
 4: s/\([[{(< ‘"_]\)\([Tt]h[eo]se\)\([]})> ’\’’",?!_.]\)/
    \1_DETERMINER_\2_\3/g
 5: s/\([[{(< ‘"_]\)\([Tt]his\)\([]})> ’\’’",?!_.]\)/
    \1_DETERMINER_\2_\3/g
 6: s/\([[{(< ‘"_]\)\([Tt]hat\)\([]})> ’\’’",?!_.]\)/
    \1_DETERMINER_\2_\3/g
 7: s/\([[{(< ‘"_]\)\([Tt]he\)\([]})> ’\’’",?!_.]\)/
    \1_DETERMINER_\2_\3/g
 8: s/\([[{(< ‘"_]\)\([Aa]n\)\([]})> ’\’’",?!_.]\)/
    \1_DETERMINER_\2_\3/g
 9: s/\([[{(< ‘"_]\)\([Aa]\)\([]})> ’\’’",?!_]\)/
    \1_DETERMINER_\2_\3/g
10: s/\([[{(< ‘"_]\)\([Aa]\)\(\.[^A-Za-z]\)/\1_DETERMINER_\2_\3/g
11: s/_TRIPLE_PERIOD_/.../g’ -  |  adjustBlankTabs -
```

In the above listing, lines 4–9 are broken at the boundary `/` of the *pattern* and the *replacement* in the `sed` substitution commands that are listed. This does not represent correct code. Line 10 shows, in principle, the “correct” code-listing for any of these `sed` substitution commands.

Explanation of the central `sed` *program:* The first substitution command (line 3) replaces the triple period as in “Bill bought...a boat and a car.” by the marker `_TRIPLE_PERIOD_`. This distinguishes the period in front of “a” in “...a boat” from an abbreviation such as “a.s.a.p.” The character preceding a determiner[10] is encoded left of the determiner in every pattern (lines 4–10) as range `[[{(< ‘"_]`, tagged and reused right[11] as `\1` in the replacement in the substitution command. The determiner which is specified in the middle of every pattern is reused as `\2`. It will be preceded by the marker `_DETERMINER_` and followed by an underscore character in the output of the above program. The non-letter following a determiner is encoded right of the determiner in the first five patterns (lines 4–8, “those”-“An”) as range `[]})> ’\’’",?!_.]`, tagged and reused as `\3`. The string `’\’’` represents a single `’` (*cf.* section 12.3.1). For the determiner “a” the period is excluded in the characters that are allowed to follow it in the range `[]})> ’\’’",?!_]` in line 9. If a period follows the character `a`, then a non-letter must follow in order that `a` represents the determiner “a”. This is encoded as `\.[^A-Za-z]` in line 10 of the program. The string

[10] This means characters that the authors consider legal to precede a word in text.
[11] That is in the continuation of the line below for lines 4–9.

encoded as `\.[^A-Za-z]` is tagged and reused as `\3`. After the tagging is completed, the triple period is restored in line 11. For example, the string `"A  liberal?"` is replaced by the program with `"_DETERMINER_A_ liberal?"`.

Application (grammatical analysis): A collection of tagging programs such as `markDeterminers` can be used for elementary grammatical analysis and search for grammatical patterns in text. If a file contains only one entire sentence per line, then a pattern `/_DETERMINER_.*_DETERMINER_/` would find all sentences that contain at least two determiners.

Note that the substitution `s/_[A-Za-z_]*_//g` eliminates everything that has been tagged thus far.

12.3.4 Turning a Text File into a Program

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

Application (removing a list of unimportant words): Suppose that one has a file that contains a list of words that are "unimportant" for some reason. Suppose in addition, that one wants to eliminate these unimportant words from a second text file. For example, function words such as *the*, *a*, *an*, *if*, *then*, *and*, *or*, ... are usually the most frequent words but carry less semantic load than content words. See [8, pp. 219–220] for a list of frequent words. The following program generates a `sed` program `$1.sed` out of a file `$1` that contains a list of words deemed to be "unimportant." The generated script `$1.sed` eliminates the unimportant words from a second file `$2`. We shall refer to the following program as `eliminateList`. For example, `eliminateList unimportantWords largeTextFile` removes words in `$1`=`unimportantWords` from `$2`=`largeTextFile`.

```
1: #!/bin/sh
2: # eliminateList
3: # First argument $1 is file of removable material.
4: # Second argument $2 is the file from which material is removed.
5: leaveOnlyWords $1  |  oneItemPerLine -  |
6: sed  's/[./-]/\\&/g
7:       s/.*/s\/\\([^A-Za-z]\\)&\\([^A-Za-z]\\)\/\\1\\2\/g/
8:      ' >$1.sed
9: addBlanks $2  |  sed -f $1.sed  -  |  adjustBlankTabs -
```

Explanation: Line 5 in the program isolates words in the file `$1` and feeds them (one word per line) into the first `sed` program starting in line 6. In the first `sed` program in lines 6–7 the following is done: 1) Periods, slashes ("A/C"), or hyphens are preceded by a backslash character. Here, the `sed`-special character `&` is used which reproduces in the replacement of a `sed` substitution command what was matched in the pattern of that command (here: the range `[./-]`). 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 the pattern space (line) containing solely the string built\-in which is matched by .* and reproduced by & in the substitution command in line 7, 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 preceded 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 to represent themselves. The list of generated s-commands is stored in a new file $1.sed in line 8. Using sed -f $1.sed in line 9, this file of s-commands is then applied to the file whose name is given to sh as second argument $2.

Application (checking summary writing): With the technique introduced above, one can compare student summaries against the original text by deleting words from the original in the students' writings.

12.3.5 Further Processing Techniques

Processing an input line repeatedly with the same (fragment of the) cycle is an important feature of sed programs and an important technique. This involves the address operator (:) and the loop operator test (t) of sed. The next example illustrates the basic mechanism in an elementary setup.

Example: The following program moves all characters 0 (zero) to the very right of a line. This shows the typical use of the t operator.

```
#!/bin/sh
# Move all zeroes to the right in a line.
sed  ': again;    s/0\([^0]\)/\10/g;    t again'  $1
```

Explanation: 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 to the left 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 (defining commutation relations and standardization for control sequences in a non plain-text file): In the course of the investigation in [1, 2, 3], techniques were developed by one of the authors to transform the file containing the source file of [37] (which was generated with a What-You-See-Is-What-You-Get Editor) 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 were used for 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 [37], Japanese words and compounds are

represented using kanji, *on* pronunciation and KUN pronunciation. The *on* pronunciation of kanji is typeset in *italics*. 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.
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, it contains a certain collection of describable irregularities. For example, the ranges of framing pairs of control sequences overlap sometimes. In order to match *on* and KUN pronunciation in the source file of [37] properly, a collection of commutation rules for control sequences was implemented such that the control sequences needed for pattern matching framed only a piece of text and no other control sequences. These commutation rules were implemented in a similar way as the latter example shows.

Application (sorting results into different files): The following example illustrates how to sort/copy results of text-processing with `sed` into a number of dedicated files. We shall refer to the following program as `sortByVowel`. `sortByVowel` 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`.

```
1: #!/bin/sh
2: # sortByVowel
3: echo >$1.a;  echo >$1.e;  echo >$1.i;  echo >$1.o;  echo >$1.u;
4: leaveOnlyWords $1  |  oneItemPerLine -  |
5: sed -n  '/a/w '$1'.a
6:          /e/w '$1'.e
7:          /i/w '$1'.i
8:          /o/w '$1'.o
9:          /u/w '$1'.u'
```

Explanation: Line 3 of this `sh` program generates empty files `$1.a`...`$1.u` in case the program has been used before on the same file. This is done by `echo`ing “*nothing* plus a terminating *newline* character” into the files.[12] First, we observe, that the option[13] `-n` (no printing) suppresses[14] any output of `sed`. Next, observe the use of the single quotes and string concatenation in `sh` in lines 5–9. For example, if the argument `$1` to `sh` equals the string `fName`, then `sh` passes the string `/a/w fName.a` to `sed` in line 5. Thus, if the current pattern space (input line) contains the vowel “a”, then `sed` writes to the file `fName.a` in line 5. Output by the `w` operator 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. (Consult `man echo` and `man rm`.) Note that everything after a `w` operator and separating white space until the end of the line is understood as the filename the `w` operator is supposed to write to. Note in addition, that a `w` operator can follow and be part of a substitution command. In that case the `w` operator writes to the named file if a substitution was made.

[12] For example, `echo 'liberal' >fName` overwrites(>) the file `fName` with the content `liberal`*newline*.

[13] When used, options are usually listed directly after a UNIX command with a leading hyphen before the first “real” argument `$1` of the command.

[14] Printing can always be triggered explicitly by the print operator `p`. For example, `/liberal/p` prints the pattern space, if the string `liberal` has been found.

There is no direct output by `sortByVowel`. It is clear how to generalize this procedure to a more significant analysis, *e.g.*, searches for specific patterns or searches for phrases.

12.4 Extending the Capabilities with awk

`awk` is a simple programming language based on pattern recognition in the current line of input and operations on chunks of that line. In that regard, it is very similar to `sed`. In contrast to `sed`, `awk` allows string-variables and numerical variables. Consequently, one can accomplish with much more ease a variety of elaborate manipulations of strings in the current line of input that may depend, in particular, on several previous lines of input. In addition, one can accomplish numerical operations on files such as accounting and keeping statistics of things (*cf.* the example `countFrequencies` in Section 12.2.2). Good introductions to `awk` are [4, 5, 30].

12.4.1 Overview of awk Programming and Its Applications

As mentioned above, `awk` is based on pattern recognition in the current line of input and operations on chunks of that line. `awk` uses pattern recognition as addresses similar to `sed` (*cf.* Section 12.3.1 and Appendix A.1). Furthermore, `awk` partitions the current line of input (input record[15]) automatically in an array of "fields". The fields of the current line of input are usually the full strings of non-white characters in the line. One typical use of `awk` is matching and rearranging the fields in a line similar to the tagging in the substitution command of `sed`. However, the tagging and reuse of tagged expressions in the substitution command of `sed` can usually only be matched by rather complicated programming in `awk`.

The Format of an awk Program

Programs in `awk` need no compilation. An `awk` program looks like the following:

```
awk  'BEGIN   { action_B } ;
pattern_1     { action_1 } ;
pattern_2     { action_2 } ;
...
END           { action_E }'
```

action$_B$ is executed before the input file is processed. *action*$_E$ is executed after the input file is processed. The lines with `BEGIN` and `END` can be omitted.

Every line in the above program contains an `awk` command (ignoring the leading string `awk  '` and the trailing `'`). One can store a list of `awk` commands in a file (say) `awkCommands` and use `awk -f awkCommands targetFile` to execute the program on `targetFile`.

`awk` operates on input records (usually lines) in a cycle just like `sed`. *action*$_1$ is executed if *pattern*$_1$ matches the original input record. After that, *action*$_2$ is executed if *pattern*$_2$ matches the current, possibly altered pattern space and the cycle was not terminated by *action*$_1$. This continues until the second-to-last line of the `awk` program

[15] This is `awk`-jargon. In `sed`-jargon, this was formerly called the pattern space.

is reached. If a *pattern*$_N$, ($N = 1, 2, ...$), is omitted, then the corresponding *action*$_N$ is executed every time the program reaches that line of code in the cycle. If `{` *action*$_N$ `}` is omitted, then the entire input line is printed by default-action. Observe that by default an `awk` program does *not* copy/print an input line (similar to `sed -n`). Thus, printing has to be triggered by an address *pattern*$_N$ with no corresponding *action*$_N$, which selects the pattern space, or alternatively, printing can be triggered by a separate `print` statement within *action*$_N$ (similar to the operator `p` in `sed`).

Example: The following program `numberOfLines` prints the number of lines[16] in a file.

```
#!/bin/sh
# numberOfLines
awk  'END {print NR}'  $1
```

`numberOfLines` prints the built-in counter `NR` (number of records) at the end of the file. By default setting, which can be changed, records are the lines of input delimited by *newline* characters. The delimiter for records is stored in the built-in variable `RS` (record separator).

Application (counting the occurrence of word-patterns in text): The following program `countIonWords` counts the occurrence of words ending in "`ion`" in a text file. Together with a search for the occurrence of words ending in "`ment`", this gives an indication of the usage of academic vocabulary in the text (*cf.* [13, 14])).

```
#!/bin/sh
# countIonWords
leaveOnlyWords $1| oneItemPerLine -| awk '/ion$/' -| numberOfLines -
```

Explanation: The first two programs of the pipe deliver one word per line into the pipe. The `awk` program[17] `/ion$/` invokes the default action `print` the pattern space (*i.e.*, line) for words ending (`$`) in "`ion`". The lines which contain such words are then counted by `numberOfLines`.

The Format of an `awk` Command

As shown above, any `awk` command has the following format:
pattern `{` *action* `} ;`
The closing semicolon is optional. If a semicolon follows an `awk` command, then another command can follow on the same line. The commands with the `BEGIN` and the `END` pattern must be on separate lines.

If *pattern* matches the input record (pattern space), then *action* is carried out. *pattern* can be very similar to address patterns in `sed`. However, much more complicated address patterns are possible in `awk`. Compare the listings in Appendix A.1 and Appendix A.2. An *action* is a sequence of statements that are separated by semicolons `;` or are on different lines.

[16] Consult the UNIX manual pages for `wc` in this regard (*i.e.*, type `man wc`).

[17] Alternatively, the `sed` program `sed '/ion$/!d' -` could be used in the pipe. `/ion$/!d` does *not* (encoded by the negation operator `!` of `sed`) *delete* (using the deletion operator `d`) a line (here: word) that ends in "`ion`". Consult also `man grep` in this regard.

Variables and Arrays

A variable-name or array-name in `awk` is a string of letters. An array-entry has the format `arrayName[`*index*`]`. The *index* in `arrayName` is simply a string which is a very flexible format, *i.e.*, any array is by default an associative array and not necessarily a linear array indexed by integers. A *typical example* for use of an associative array showing the power of this concept can be found in Section 12.2.2 of this chapter (`countFrequencies`). Numbers are simultaneously understood as strings in `awk`. All variables or entries of an array that are used are automatically initiated to the empty string. Any string has numerical value zero (0).

Built-In Variables

`awk` has a number of built-in variables some of which have already been introduced. In the next few paragraphs, we shall list the most useful ones. The reader is refered to [4, 5, 40] or the UNIX manual pages for `awk` for a complete listing.

`FILENAME`: The built-in variable `FILENAME` contains the name of the current input file. `awk` can distinguish the standard input `-` as the name of the current input file. Using a pattern such as `FILENAME==fName` (*cf.* appendix A.2), processing by `awk` can depend upon one of several input files that follow the `awk` program as arguments and are being processed in the order listed from left to right (*e.g.*, `awk '`*awkProgram*`' fileOne fileTwo fileLast`). See the listing of the program `setIntersection` below in Section 12.4.2.3 for a typical use of `FILENAME`.

`FS`: The built-in variable `FS` contains the field separator character. Default: sequences of blanks and tabs. For example, the variable `FS` should be reset to `&` (separator for tables in TEX), if one wants to partition the input line in regard to fields separated by `&`. Such a resetting action happens often in $action_B$ matched by the `BEGIN` pattern at the start of processing in an `awk` program.

`NF`: The built-in variable `NF` contains the number of fields in the current pattern space (input record). This is very important in order to loop over all fields in the pattern space using the `for`-loop construct of `awk`. A typical loop is given by:

`for(counter=1;counter<=NF;counter++){` *actionWith*`(counter) }`.

See the listing of the program `findFourLetterWords` below for a typical use of `NF`. 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`: The built-in variable `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. See the listing of the program `context` below for a typical use of `NR`.

`OFS`: The built-in variable `OFS` contains the output field separator used in `print`. Default: blank. `OFS` is caused to be printed if a comma "`,`" is used in a `print` statement. See the listing of the program `firstFiveFieldsPerLine` below for an application.

`ORS`: The built-in variable `ORS` contains the output record separator string. It is appended to the output after each `print` statement. Default: *newline*-character. `ORS` can be set to the empty string through `ORS=""`. In that case, output lines are concatenated. If one sets `ORS="\n\n"`, *i.e.*, two newline characters (see next section), then the output is double-spaced. See the listing of the `awk` program in section 12.5.2 for an application.

`RS`: The built-in variable `RS` contains the input record separator character. Default: *newline* character. Note that one can set `RS="\n\n"`. In that case, the built-in variable `NR` counts paragraphs, if the input text file is single-spaced.

Representation of Strings, Concatenation and Formatting the Output

Strings of characters in `awk` used in printing and as constant string-values are simply framed by double quotes `"`. The special character sequences `\\`, `\"`, `\t` and `\n` represent the backslash, the double quote, the tab and the newline character in strings respectively. Otherwise, every character including the blank just represents itself.

Strings or the values of variables containing strings are concatenated by listing the strings or variables separated by blanks. For example, `"aa" "bb"` represents the same string as `"aabb"`.

A *string* (framed by double quotes `"`) or a variable `var` containing *string* can be printed using the statements '`print` *string*`;`' or '`print var;`' respectively. The statement `print;` simply prints the pattern space. Using the `print` function for printing is sufficient for most purposes. However in `awk`, one can also use a second printing function `printf` which acts similar to the function `printf` of the programming language C. See [40, 31, 4, 5] for further details and consult the manual pages for `awk` and `printf` for more information on `printf`. One may be interested in `printf` if one wants to print the results of numerical computations, such as statistical evaluations for further processing by a plotting program such as Mathematica [47] or `gnuplot` [17].

Application (finding a line together with its predecessor in a text): 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 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 precede such lines. We shall refer to it as `printPredecessorBecause`.

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

Explanation: The symbol/string `$0` represents the entire line or pattern space in `awk`. Thus, if the current line matches `/[Bb]ecause/`, then it is printed following its predecessor which was previously saved in the variable `previousLine`. Afterwards, two *newline* characters are printed in order to structure the output. Should the first line of the input file match `/[Bb]ecause/`, then `previousLine` shall be automatically initiated to the empty string such that the output starts with the first *newline* character that is printed. Finally, every line is saved in the variable `previousLine` waiting for the next input line and cycle.

Fields and Field Separators

In the default mode, the fields of an input line are the full strings of non-white characters separated by blanks and tabs. They are addressed in the pattern space from left to right as field variables `$(1)`, `$(2)`, ... `$(NF)` where `NF` is a built-in variable containing the number of fields in the current input record. Thus, one can loop over all fields in the current input record using the `for`-statement of `awk` and manipulate every field separately. Alternatively, `$(1)`—`$(9)` can be addressed as `$1`—`$9`. The symbols/strings `$0` and `$(0)` stand for the entire pattern space.

Example: The following program `firstFiveFieldsPerLine` prints the first five fields in every line separated by one blank. It can be used to isolate starting phrases of sentences, if a text file is formatted in such a way that every line contains an entire single sentence. For example, it enables an educator to check whether his or her students use transition signals such as "`First`", "`Next`", "`In short`" or "`In conclusion`" in their writing.

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

Recall that the trailing `$1` represents the input file name for the Bourne shell. The commas trigger printing of the built-in variable `OFS` (output field separator) which is set to a blank by default.

Built-In Operators and Functions

`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 parentheses.

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

- Increment operators `++`, `--`. Comment: `++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).
- Algebraic operators `*`, `/`, `%`. Comment: Multiplication, division, and integer division remainder (mod-operator).
- Concatenation of strings. Nothing or white space (*cf.* Section 12.4.1.5).
- Relational operators for comparison `>`, `>=`, `<`, `<=`, `==`, `!=`, `~`, `!~`. Comment: `==`, `!=` stand for "equal" and "not equal," respectively. `~`, `!~` stand for "matches pattern" and "does not match pattern," respectively. For example, `x~/a/` is satisfied, if the string in variable `x` contains the letter `a`. If it is not clear what sort of comparison is meant, then `awk` uses string comparison instead of numerical comparison.
- `!`. Logical NOT.
- `&&`. Logical AND.
- `||`. Logical OR.
- Assignment operators `=`, `+=`, `-=`, `*=`, `/=` and `%=`. Comment: `=` is the assignment operator that assigns a value to a variable. The other assignment operators `+=`, `-=`, `*=`, `/=` and `%=` exist just for notational convenience as, *e.g.*, in C [31]. For example, `var+=d` sets `var` to `var+d`. This the same as `var=var+d`.

In addition to the above operators, the following built-in functions can be used in `awk` programs:

- `int`, `sqrt`, `exp`, `log`. Comment: `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.

• `length(`*string*`)` returns the length of *string*, *i.e.*, the number of characters in *string.*

• `index(`*bigstring*`,`*substring*`)`. Comment: 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`)`. Comment: This produces the n_1^{th} through 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.
• `split(`*string*`,arrayName,"c")`. Comment: This splits *string* at every instance of the separator character `c` into the array `arrayName` and returns the number of fields encountered.
• `string = sprintf(`*format* , *expr1* , *expr2 ...*`)`. Comment: This sets `string` to what is produced by `printf` *format* , *expr1* , *expr2 ...* In regard to the printing function `printf` in `awk` or C consult [40, 31, 4, 5] and the manual pages for `awk` and `printf`.

Application (generating strings of context from a file): The next important example shows the use of the functions `index()` and `substr()` in `awk`. It generates all possible sequences of consecutive words of a certain length in a file. 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` of consecutive words in `$1` are generated and printed.

```
1: #!/bin/sh
2: # context
3: # First argument $1 is input file name.
4: # Second argument $2 is context-length.
5: awk  'BEGIN { cLength='$2'+0 }
6: NR==1       { c=$0                      }
7: NR>1        { c=c " " $0 }
8: NR>cLength  { c=substr(c,index(c," ")+1) }
9: NR>=cLength { print c          }'  $1
```

Explanation: Suppose the above program is invoked as `context sourceFile 11`. Then, `$2`=11. In line 5, the `awk`-variable `cLength` is set to 11. Thereby, the operation `+0` forces any string contained in the second argument `$2` to `context`, even the empty string, to be considered as a number in the remainder of the program. In the second command of the `awk` program (line 6), the context `c` is set to the first word (*i.e.*, input line). In the third command (line 7), any subsequent word (input line) other than the first is appended to `c` separated by a blank. The fourth statement (line 8) 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. Thus, the word at the very left of `c` is lost. Finally (line 9), the context `c` is printed, if it contains at least 11 words `cLength`.

Note that the output of `context` is, 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.

Control Structures

`awk` has two special control structures `next` and `exit`. In addition, `awk` has the usual control structures: `if`, `for` and `while`.

`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 `d` operator in `sed`. `exit` is a statement that causes `awk` to terminate immediately. `exit` is the analogue of the `q` operator in `sed`.

The `if` statement looks the same as in C [31, p. 55]:
`if (`*conditional*`) {` *action1* `}`
`else {` *action2* `}`
conditional can be any of the types of conditionals we defined above for address patterns including Boolean combinations of comparison of algebraic expressions including the use of variables. If a regular expression `/`*regExpr*`/` is intended to match or not to match the entire pattern space `$0` in *conditional*, then this has to be denoted explicitely using the match-operator `~`. Thus, one has to use `$0~/`*regExpr*`/` or `$0!~/`*regExpr*`/` respectively. The `else` part of the `if` statement can be omitted or can follow on the same line.

Example: The use of a `for`-statement in connection with an `if`-statement is shown in the next example. We shall refer to the following program as `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.

```
1: #!/bin/sh
2: # findFourLetterWords (awk version)
3: leaveOnlyWords $1  |
4: awk '       { for(f=1;f<=NF;f++) {
5:                 if($(f)!~/^[A-Za-z][a-z][a-z][a-z]$/) { $(f)="" }
6:               }
7:             }
8:      /[^ ]/ { print NR , $0 }
9:     ' -  |  adjustBlankTabs -
```

Explanation: The `for`-loop in line 4 processes every field (addressed as `$(f)` in line 5, `f` the counter variable) from left to right. If a field `$(f)` does not match the pattern `/^[A-Za-z][a-z][a-z][a-z]$/`, then it is set to the empty string in line 5. In case the pattern space stays non-white (`/[^ ]/`) after this procedure, it is printed in line 8 with a leading line-number `NR`. Finally, blanks are properly adjusted in the output by `adjustBlankTabs`.

The technique how to loop over associative arrays has already been demonstrated in the listing of the program `countFrequencies` in Section 12.2.2.

Similar to the `for` statement, the `while` statement also looks the same as in `C` [31, p. 60]: `while (` *conditional* `) {` *action* `}`.

12.4.2 Vectors and Sets

We conclude the section on awk by introducing a standard file format called "vectors." For files of this format, we show how to define a large variety of operations such as vector addition/subtraction and statistical operations. In addition, we define set-operations. Such operations are very useful in numerical/statistical evaluations and for comparison of data obtained by methods presented until this point in our exposition.

Definition: Vector (Lists of Type/Token Ratios)

Suppose that one represents frequencies of occurrence of particular words or phrases in the following way in a file: every line of the file consists of two parts where the first part is a word or phrase which may contain digits and the second part (the final field) is a single number which represents and will be called the frequency. A file in this format will be called a *vector* (*list of type/token ratios*). An example of an entry of a vector is given by

```
limit  55
```

Mathematically speaking, such a file of word/phrase frequencies is a vector over the free base of character strings [20, p. 13]. The program countFre- quencies listed in Section 12.2.2 generates vectors.

Vector Operations

In this section, we show how to implement vector operations using awk.

Application (vector addition): The next program vectorAddition implements vector addition. If aFile and bFile are vectors, then vectorAddition is used as cat aFile bFile | vectorAddition -. The UNIX command cat aFile bFile concatenates files aFile bFile with the content of aFile leading.

vectorAddition can be used, *e.g.*, to measure the cumulative advance of students in regard to vocabulary use.

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

Explanation: 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 sum up (+=) all frequencies n corresponding to the altered string $0. Recall that sum[$0] is initiated to 0 automatically. After processing the input this way (at the END), the for-loop passes through the associative array sum with looping index string. string is printed together with the values of the summations (sum[string]). Note that there is no comma in the print statement in view of (*). Finally, the overall output is sorted into standard lexicographical order using the UNIX command sort.

Application (scalar multiplication): The next program `scalarMultiplica- tion` implements scalar multiplication. If `aFile` is a vector and `n` is a number, then it is used as `scalarMultiplication aFile n`.

```
#!/bin/sh
# scalarMultiplication
# First argument $1 is vector. Second argument $2 is scalar.
awk  '{  $(NF)*=('$2'+0) ;  print  }'  $1
```

Explanation: The scalar `$2` is spliced into the `awk` program by `sh` using string concatenation of the strings `'{ $(NF)*=('`, the content of the second argument of the command `$2` and `'+0) ; print }'`. Then, for every input line of the file `$1`, every frequency in the vector which is stored in `$(NF)` is multiplied by the scalar and the resulting pattern space is printed.

Application (absolute value): The next program `computeAbsoluteValue` computes the absolute value of the frequencies of items in a vector.

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

Explanation: If the last field of an input line is negative, then its sign is reversed. Next, every line is printed.

Application (sign function): Like the previous program, the next program `frequencySign` computes the sign of the frequencies of items in a vector.

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

Application (selecting frequencies): The next program cuts away low frequencies from a file `$1` that is a vector. The limit value `$2` is the second argument to the program. We shall refer to it as `filterHighFrequencies`. 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
# filterHighFrequencies
# First argument $1: vector. Second argument $2: cut-off threshold.
awk  '$(NF)>='$2  $1
```

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

Application: The vector operations presented above allow to analyse and compare, *e.g.*, vocabulary use of students in a class in a large variety of ways (vocabulary use of a single student *vs.* the class or *vs.* a dedicated list of words, similarity/distinction of vocabulary use among students, computation of probalility distributions over vocabulary use (normalization), etc.).

Application (average and standard deviation): The following program determines the sum, average and standard deviation of the frequencies in a vector `$1`.

```
#!/bin/sh
awk  '/[^  ]/ {  s1+=$(NF);  s2+=$(NF)*$(NF)               }
      END     {  print s1 ,  s1/NR ,  sqrt(s2*NR-s1*s1)/NR  }'  $1
```

Explanation: The `awk` program only acts on non-white lines since the non-white pattern `/[^  ]/` must be matched. `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 $\texttt{s1}=\sum_{n=1}^{\texttt{NR}} \texttt{\$(NF)}_n$ and $\texttt{s2}=\sum_{n=1}^{\texttt{NR}} (\texttt{\$(NF)}_n)^2$. In the `END`-line, the sum `s1`, the average `s1/NR` and the standard deviation (*cf.* [16, p. 81]) are printed.

Set Operations

In this section, we show how to implement set operations using `awk`. Set operations as well as vector operations are extremely useful in comparing results from different analyses performed with the methods presented thus far.

Application (set intersection): The next program implements set intersection.[18] We shall refer to it as `setIntersection`. If `aFile` and `bFile` are organized such that items (= set elements) are listed on separate lines, then it is used as `setIntersection aFile bFile`. `setIntersection` can be used to measure overlap in use of vocabulary. Consult also `man comm`.

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

Explanation: `awk` can accept and distinguish more than one input file after the program-string. This property is utilized here. 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*.

If one changes the final conditional `n[$0]==1` in `setIntersection` to `n[$0]==0`, then this implements set-complement. If such a procedure is named `setComplement`, then `setComplement aFile bFile` computes all elements from `bFile` that are not in `aFile`.

[18] Note that `adjustBlankTabs fName | sort -u -` converts any file `fName` into a set where every element occurs only once. In fact, `sort -u` sorts a file and only prints occurring lines once. Consequently, `cat aFile bFile | adjustBlankTabs - | sort -u -` implements set union.

12.5 Larger Applications

In this section, we describe how these tools can be applied in language teaching and language analysis. We draw on our experience using these tools at the University of Aizu where Japanese students learn English as a foreign language. Of course, any language teacher can modify the examples outlined here to fit a given teaching need.

The tools provide three types of assistance to language teachers: they can be used for teaching, for language analysis to inform teaching, and for language analysis in research. In teaching, the tools can be linked to an email program that informs students about specific errors in their written texts; such electronic feedback for low-level errors can be more effective than feedback from the teacher [43, 45]. The tools can also help teachers identify what needs to be taught. From a database of student writing, a teacher can identify systematic patterns of errors that need to be addressed in class. In addition, one can isolate syntactic structures and lexical items that the students either overuse or avoid using because of their complexity [27].

One can also use the tools to identify (or confirm) the features of expert texts in different research genres. Such texts are organized along similar lines with four sections– Introduction, Methods, Results, and Discussion; further, the Introduction section can be divided into four Moves [42] that use different language structures and lexis. Other features include the location of the thesis sentence [7], the use of hedges such as “perhaps” and “could” [24], and the use of cohesive devices such as repetition to make the text more readable [22]. Since most students are not familiar with such devices, the teacher may need to examine expert texts and use the `awk` and `sed` tools to locate similar strings in student writing.

Various commercial tools are currently available for language analysis; however, many of them come in separate packages and are often expensive. Further, they draw on million-word databases that provide accurate results but are overkill for both teachers and students.

The following examples illustrate some of the capabilities of the techniques developed thus far.

12.5.1 Automated Feedback for Spelling and Punctuation

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 English, and mechanical mistakes are abundant.

Spelling errors can be identified with the UNIX `spell` program. In [39], we use `sed` and `awk` to reformat the result of the spell check, which is sent back to the student.

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 manually correcting persistent errors 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-correction program such as the one in [39] described below is able to correct nearly all of the students' punctuation problems, thus presenting the spacing rules in an inductive, interactive way. A punctuation-correcting program is one of several tools described in [35].

As a database, we have defined a list of forbidden pairs of characters. This is achieved by listing the matrix M pertaining to the relation R which is given by $\mathtt{char}_1\ R\ \mathtt{char}_2 \Leftrightarrow$ "The character sequence $\mathtt{char}_1\mathtt{char}_2$ is forbidden." During the setup phase of the system used in [39], the matrix M is translated by an `sed` program into a new `sed` program which scans the essays submitted by students via electronic mail for mistakes. Examples for forbidden sequences are *blank*`,` or ‘?. These mistakes are marked, and the marked essays are sent 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 in Section 12.3.4. 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 [39].

Finally, let us remark that our procedure for identifying mistakes in punctuation can also be used in analyses of punctuation patterns, frequency, and use, as in [36].

12.5.2 Extracting Sentences

In [39], one of the tools reformats student essays in such a way that entire sentences are on single lines. Such a format is very useful in two ways:
Goal 1: To select actual student sentences which match certain patterns. The teacher can then write any number of programs that search for strings identified as particularly problematic for a given group of students. For example, the words "because" and "too" are frequently used incorrectly by Japanese speakers of English. 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 for 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 [26].

The following shows part of the implementation of sentence identification in [39]:

```
#!/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.

```
1: sed 's/\([^]})'\''".!?][]}).!?]*\)\([!?]\)
   \([]})]*\)\([^]})'\''".!?]\)/\1\2\3__\2__\
2: \4/g
3: s/\([^]})'\''".!?][]}).!?]*\)\([!?]\)\([]})]*\)$/\1\2\3__\2__/
4: s/\([^]})'\''".!?][]})'\''".!?]*\.
   []})'\''"]*\)\([^]})'\''".!?]\)/\1__.__\
5: \2/g
6: s/[^]})'\''".!?][]})'\''".!?]*\.[]})'\''"]*$/&__.__/' |
```

Explanation: Line 1 of this listing is broken after `\([!?]\)` representing the end of the sentence . In the first two `sed` commands (lines 1–3), the end of the sentence for "?" and "!" are defined. The similar treatment of "?" and "!" is implemented by using a range `[!?]` which is the second tagged entity in the patterns in lines 1 and 3. Thus, the letter ending the sentence is represented by `\2`. The range-sequence `[^]})'\''".!?]` followed by `[]}).!?]*` defines admissible strings before the end of a sentence. It is the first tagged entity `\1` in the patterns in lines 1 and 3. The range-sequence represents at least one non-closing character, followed by a possible sequence of allowed closing characters. A sentence may be multiply bracketed in various ways. This is represented by the range `[]})]*` which is the third tagged entity `\3` in the patterns in lines 1 and 3. After the possible bracketing is finished, there should not follow another closing (brackets, quotes) or terminating character ".", "?" or "!". (This handles exactly the case of the previous sentence.) The excluded terminating character is encoded as `[^]})'\''".!?]` in line 1, and is the fourth tagged item `\4`. In the substitution part of the `sed` command in lines 1–2, the originally tagged sequence (`\1\2\3\4`) is replaced by `\1\2\3__\2__`*newline*`\4`. Thus, after the proper ending of the sentence in `\3`, a marker `__\2__` is introduced for sorting/identification purposes. Then, a *newline* character is introduced such that the next sentence starting in `\4` starts on a new line. Line 3 handles the case when the sentence-end in "?" or "!" coincides with the end of the line.

Line 4 of this listing is broken after `\.` representing the period (and not an arbitrary character) ending the sentence. The last two substitution rules in lines 4–6 for marking sentences that end in a period are different than those for "?" and "!". But the principles are similar. In line 4, the range-sequence `[^]})'\''".!?][]})'\''".!?]` followed by `[]})'\''".!?]*` defines admissible strings before the end of a sentence. The range-sequence represents at least one non-closing character, followed by a possible sequence of allowed closing characters. Then the closing period is explicitly encoded as `\.` The range-sequence `[]})'\''"]*` (closing brackets) followed by `[^]})'\''".!?]` (non-closing character) defines admissible strings after the end of a sentence. Line 7 handles the case when the sentence-end coincides with the end of the line.

Next follows an `awk` program in the pipe which is shown below:

```
awk 'BEGIN   { ORS=" " }
             { print }
```

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

Explanation: The program merges lines that are not marked as sentence endings by setting the output record separator `ORS` to a blank. If a line-end is marked as sentence-end, then an extra newline character is printed.

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.

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 `printPredecessorBecause`, it is clear how to connect the output of the sentence finder with a program that searches for `and so on`.

In [46], 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.

12.5.3 Readability of Texts

Hoey [22, pp. 35–48, 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 "rough and ready analysis" [22, p. 235] by hand is tedious work, and 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 (*i.e.*, within a string of 200 words) 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 [22, pp. 35–48].

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

Explanation: `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. An implementation of `quadrupleWords` is left as an exercise; or consult [40]. `countFrequencies` determines the word frequencies of the determined words.

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

We have applied the above procedure to the source file of an older version of this document. Aside from function-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.*

12.5.4 Lexical-Etymological Analysis

In [19], the author determined the percentage of etymologically related words shared by Serbo-Croatian, Bulgarian, Ukrainian, 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 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 written in an alphabetic writing system into a computer to obtain automatically a file format that can be evaluated by 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.* [15]).

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.

12.5.5 Corpus Exploration and Concordance

The following `sh` program shows how to generate the surrounding context for words from a text file `$1`, *i.e.*, the file name is first argument `$1` to the program. The second argument to the program, *i.e.*, `$2`, is supposed to be a strictly positive integer. In this example, two words are related if there are not more that (`$2`)−2 other words in between them.

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

Explanation: If a file contains the strings (words) `aa`, `ab`, `ac`, ... `zz` and `$2`=6, then the first line of output of the code in lines 3–4 (into the pipe continued at line 5) would be `aa ab ac ad ae af`. That is what the `awk` program in line 5 would see as first line of input. The `awk` program would then print `aa ab`, `aa ac`, `...` `aa af` on separate lines as response to that first line of input. The occurrence of such pairs is then counted by `countFrequencies`. This defines a matrix M_d of directed context (asymmetric relation) between the words in a text. M_d is indexed by pairs of words ($word_1$,$word_2$). If the frequency of the entry in M_d pertaining to ($word_1$,$word_2$) is low, then the two words $word_1$ and $word_2$ are distant or unrelated.

Applying the procedure listed above to the source file of an older version of this document and filtering out low frequencies using `filterHighFrequencies - 20` the following pairs of word were found in close proximity 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. See the example given in the introduction.

In [38], a corpus search for the strings characterized by the following `awk` patterns

```
(a|(an)|(for)|(had)|(many)) [A-Za-z'-]+ of
((be)|(too)) [A-Za-z'-]+ to
```

was conducted. Modifying the program listed in the introduction 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") {
                        File="'$1'." $(11) ".of"; print>File }
     ($(11)~/^((be)|(too))$/)              &&($(13)=="to") {
                         File="'$1'." $(11) ".to"; print>File }' -
```

It has been noted in several corpus studies of English collocation ([32, 41, 6]) 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 [12, 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 [9], 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 ((mine)|(yours)|...)` (*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.

12.5.6 Reengineering Text Files across Different File Formats

In the course of the investigations outlined in [1, 2, 3], one of the authors developed a family of programs that are able to transform the source file of [37], 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 slightly altering the programs already developed.

The source was available in two formats: 1) an RTF format file, and 2) a text file free of control sequences that was generated from the first file. Both formats have advantages and disadvantages. As outlined in Section 12.3.5, 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 outlined in Section 12.3.5 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, which is useful in regard to pattern matching that identifies the three different categories of entries in [37]: *radical*, *kanji* and *compound*. However, very valuable information is lost in the second file in regard to the distinction between *on* pronunciation, KUN pronunciation and English.

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` operator `l`[19]. As outlined in Section 12.3.5, 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 [37] is typed regularly enough such that the three categories of entry *radical*, *kanji* and *compound* can be identified using pattern matching. In fact, a small grammar was defined for the structure of the source file of [37] and verified with `awk`. By simply counting all units, an index for the dictionary which does not exist in [37] can now be generated. 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. It is also easy to generate, *e.g.*, English→*kanji* or English→KUN dictionaries from this *kanji*→*on*/KUN→English dictionary using the UNIX command `sort` and rearrangement of fields. In addition, it is easy to reformat [37] into proper `jlatex` format. This could be used to re-typeset the entire dictionary.

12.6 Conclusion

In the previous exposition, we have given a short but detailed introduction to `sed` and `awk` and their applications to language analysis. 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 create 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 [39]. 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 an entire commercial programming package. We have shown how to duplicate in a simple way searches performed by large third-party packages. 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

[19] The `sed` operator `l` lists the pattern space on the output in an unambiguous form. In particular, non-printing characters are spelled in two-digit ASCII and long lines are folded.

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 very efficient.

Disclaimer

The authors do not accept responsibility for any line of code or any programming method presented in this work. There is absolutely no guarantee that these methods are reliable or even function in any sense. Responsibility for the use of the code and methods presented in this work lies solely in the domain of the applier/user.

References

1. H. Abramson, S. Bhalla, K.T. Christianson, J.M. Goodwin, J.R. Goodwin, J. Sarraille (1995): Towards CD-ROM based Japanese $\leftrightarrow$ English dictionaries: Justification and some implementation issues. In: *Proc. 3rd Natural Language Processing Pacific-Rim Symp.* (Dec. 4–6, 1995), Seoul, Korea
2. H. Abramson, S. Bhalla, K.T. Christianson, J.M. Goodwin, J.R. Goodwin, J. Sarraille, L.M. Schmitt (1996): Multimedia, multilingual hyperdictionaries: A Japanese $\leftrightarrow$ English example. Paper presented at the *Joint Int. Conf. Association for Literary and Linguistic Computing and Association for Computers and the Humanities* (June 25–29, 1996), Bergen, Norway, available from the authors
3. H. Abramson, S. Bhalla, K.T. Christianson, J.M. Goodwin, J.R. Goodwin, J. Sarraille, L.M. Schmitt (1996): The Logic of Kanji lookup in a Japanese $\leftrightarrow$ English hyperdictionary. Paper presented at the *Joint Int. Conf. Association for Literary and Linguistic Computing and Association for Computers and the Humanities* (June 25–29, 1996), Bergen, Norway, available from the authors
4. A.V. Aho, B.W. Kernighan, P.J. Weinberger (1978): awk — A Pattern Scanning and Processing Language (2nd ed.). In: B.W. Kernighanm, M.D. McIlroy (eds.), *UNIX programmer's manual (7th ed.)*, Bell Labs, Murray Hill, `http://cm.bell-labs.com/7thEdMan/vol2/awk`
5. A.V. Aho, B.W. Kernighan, P.J. Weinberger (1988): *The AWK programming language*. Addison-Wesley, Reading, MA
6. B.T.S. Atkins (1992): *Acta Linguistica Hungarica* **41**:5–71
7. J. Burstein, D. Marcu (2003): *Computers and the Humanities* **37**:455–467

8. C. Butler (1985): *Computers in linguistics.* Basil Blackwell, Oxford
9. K.T. Christianson (1997): *IRAL* **35**:99–113
10. K. Church (1990): Unix for Poets. Tutorial at *13th Int. Conf. on Computational Linguistics, COLING-90* (August 20–25, 1990), Helsinki, Finland, `http://www.ling.lu.se/education/homepages/LIS131/unix_for_poets.pdf`
11. W.F. Clocksin, C.S. Mellish (1981): *Programming in Prolog.* Springer, Berlin
12. A. Collier (1993): Issues of large-scale collocational analysis. In: J. Aarts, P. De Haan, and N. Oostdijk (eds.), *English language corpora: Design, analysis and exploitation*, Editions Rodopi, B.V., Amsterdam
13. A. Coxhead (2000): *TESOL Quarterly* **34**:213–238
14. A. Coxhead (2005): Academic word list. Retrieved Nov. 30, 2005, `http://www.vuw.ac.nz/lals/research/awl/`
15. A. Fox (1995): *Linguistic Reconstruction: An Introduction to Theory and Method.* Oxford Univ. Press, Oxford
16. P.G. Gänssler, W. Stute (1977): *Wahrscheinlichkeitstheorie.* Springer, Berlin
17. GNUPLOT 4.0. Gnuplot homepage, `http://www.gnuplot.info`
18. J.D. Goldfield (1986): An Approach to Literary Computing in French. In: *Méthodes quantitatives et informatiques dans l'étude des textes*, Slatkin-Champion, Geneva
19. M. Gordon (1996): What does a language's lexicon say about the company it keeps?: A slavic case study. Paper presented at *Annual Michigan Linguistics Soc. Meeting* (October 1996), Michigan State Univ., East Lansing, MI
20. W. Greub (1981): *Linear Algebra.* Springer, Berlin
21. S. Hockey, J. Martin (1988): *The Oxford concordance program: User's manual (Ver. 2).* Oxford Univ. Computing Service, Oxford
22. M. Hoey (1991): *Patterns of lexis in text.* Oxford Univ. Press, Oxford
23. A.G. Hume, M.D. McIlroy (1990): *UNIX programmer's manual (10th ed.).* Bell Labs, Murray Hill
24. K. Hyland (1997): *J. Second Language Writing* **6**:183–205
25. S.C. Johnson (1978): Yacc: Yet another compiler-compiler. In: B.W. Kernighan, M.D. McIlroy (eds.), *UNIX programmer's manual (7th ed.)*, Bell Labs, Murray Hill, `http://cm.bell-labs.com/7thEdMan/vol2/yacc.bun`
26. G. Kaye (1990): A corpus builder and real-time concordance browser for an IBM PC. In: J. Aarts, W. Meijs (eds.), *Theory and practice in corpus linguistics*, Editions Rodopi, B.V., Amsterdam
27. P. Kaszubski (1998): Enhancing a writing textbook: a nationalist perspective. In: S. Granger (ed.), *Learner English on Computer*, Longman, London
28. G. Kennedy (1991): *Between* and *through*: The company they keep and the functions they serve. In: K. Aijmer, B. Altenberg (eds.), *English corpus linguistics*, Longman, New York
29. B.W. Kernighan, M.D. McIlroy (1978): *UNIX programmer's manual (7th ed.).* Bell Labs, Murray Hill
30. B.W. Kernighan, R. Pike (1984): *The UNIX programming environment.* Prentice Hall, Englewood Cliffs, NJ
31. B.W. Kernighan, D.M. Ritchie (1988): *The C programming language.* Prentice Hall, Englewood Cliffs, NJ
32. G. Kjellmer (1989): Aspects of English collocation. In: W. Meijs (ed.), *Corpus linguistics and beyond*, Editions Rodopi, B.V., Amsterdam
33. L. Lamport (1986): *Latex — A document preparation system.* Addison-Wesley, Reading, MA

34. M.E. Lesk, E. Schmidt (1978): Lex — A lexical analyzer generator. In: B.W. Kernighan, M.D. McIlroy (eds.), *UNIX programmer's manual (7th ed.)*, Bell Labs, Murray Hill, `http://cm.bell-labs.com/7thEdMan/vol2/lex`
35. N.H. McDonald, L.T. Frase, P. Gingrich, S. Keenan (1988): *Educational Psychologist* **17**:172–179
36. C.F. Meyer (1994): Studying usage in computer corpora. In: G.D. Little. M. Montgomery (eds.), *Centennial usage studies*, American Dialect Soc., Jacksonville, FL
37. A.N. Nelson (1962): *The original modern reader's Japanese-English character dictionary (Classic ed.).* Charles E. Tuttle, Rutland
38. A. Renouf, J.M. Sinclair (1991): Collocational frameworks in English. In: K. Aijmer, B. Altenberg (Eds.) *English corpus linguistics*, Longman, New York
39. L.M. Schmitt, K. Christianson (1998): *System* **26**:567–589
40. L.M. Schmitt, K. Christianson (1998): ERIC: Educational Resources Information Center, Doc. Service, National Lib. Edu., USA, **ED** 424 729, **FL** 025 224
41. F.A. Smadja (1989): *Literary and Linguistic Computing* **4**:163–168
42. J.M. Swales (1990): *Genre Analysis: English in Academic and Research Setting.* Cambridge Univ. Press, Cambridge
43. F. Tuzi (2004): *Computers and Composition* **21**:217–235
44. L. Wall, R.L. Schwarz (1990): *Programming perl.* O'Reilly, Sebastopol
45. C.A. Warden (2000): *Language Learning* **50**:573–616
46. J.H.M. Webb (1992): 121 *common mistakes of Japanese students of English (Revised ed.).* The Japan Times, Tokyo
47. S. Wolfram (1991): *Mathematica — A system for doing mathematics by computer (2nd ed.).* Addison-Wesley, Reading, MA

Appendices

A.1. Patterns (Regular Expressions)

Patterns which are also called regular expressions can be used in `sed` and `awk` for two purposes:

(a) As addresses, in order to select the pattern space (roughly the current line) for processing (*cf.* sections 12.3.1 and 12.4.1).

(b) As patterns in `sed` substitution commands that are actually replaced. Patterns are matched by `sed` and `awk` as the *longest, non-overlapping strings* possible.

Regular expressions in `sed`. The patterns that can be used with `sed` consist of the following elements in between slashes `/`:

(1) Any non-special character matches itself.

(2) Special characters that otherwise have a particular function in `sed` have to be preceded by a backslash `\` in order to be understood literally. The special characters are: `\\ \/ \^ \$ \. \[ \] \* \& \n`

(3) `^` resp. `$` match the beginning resp. the end of the pattern space. They must not be repeated in the replacement in a substitution command.

(4) `.` matches any single character.

(5) `[`*range*`]` matches any character in the string of characters *range*. The following five rules must be observed:
R1: The backslash `\` is not needed to indicate special characters in *range*. The backslash only represents itself.
R2: The closing bracket `]` must be the first character in *range* in order to be recognized as itself.
R3: Intervals of the type `a-z`, `A-Z`, `0-9` in *range* are permitted. For example, `i-m`.
R4: The hyphen `-` must be at the beginning or the end of *range* in order to be recognized as itself.
R5: The carat `^` must not be the first character in *range* in order to be recognized as itself.

(6) `[^`*range*`]` matches any character not in *range*. The rules *R1–R4* under 5) also apply here.

(7) *pattern*`*` stands for 0 or any number of concatenated copies of *pattern* where *pattern* is a specific character, the period `.` (meaning any character) or a range `[...]` as described under 5) and 6).

(8) *pattern*`\{`α`,`ω`\}` stands for α to ω concatenated copies of *pattern*. If ω is omitted, then an arbitrarily large number of copies of *pattern* is matched. Thus, the repitor `*` is equivalent to `\{0,\}`.

Regular expressions in `awk`. Regular expressions 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. Regular expressions in `awk` are very similar to regular expressions in `sed`. The regular expressions that can be used with `awk` consist of the following elements in between slashes `/`:

(1) Any non-special character matches itself as in `sed`.

(2) Special characters that otherwise have a particular function in `awk` have to be preceded by a backslash `\` in order to be understood literally as in `sed`. A *newline* character in the pattern space can be matched with `\n`. The special characters are: `\\ \/ \^ \$ \. \[ \] \* \+ \? \( \) \| \n`.

Observe that `&` is not special in `awk` but in `sed`. In contrast, `+` and `?` are special in `awk` serving as repitors similar to `*`. Parentheses are allowed in regular expressions in `awk` for grouping. Alternatives in regular expressions in `awk` are encoded using the vertical slash character `|`. Thus, the literal characters `\+`, `\?`, `\(`, `\)` and `\|` become special in `awk` but are not in `sed`. Note that there is no tagging using `\(` and `\)` in `awk`.

(3) `^` resp. `$` match the beginning resp. the end of the pattern space as in `sed`.

(4) `.` matches any single character as in `sed`.

(5) `[`*range*`]` matches any character in the string of characters *range*. The following five rules must be observed:
R1: The backslash `\` is not used to indicate special characters in *range* except for `\]` and `\\`.
R2: The closing bracket `]` is represented as `\]`. The backslash `\` is represented

as \\.
R3: Intervals of the type `a-z`, `A-Z`, `0-9` in *range* are permitted. For example, `1-9`.
R4: The hyphen `-` must be at the beginning or the end of *range* in order to be recognized as itself.
R5: The carat `^` must not be the first character in *range* in order to be recognized as itself.

(6) `[^`*range*`]` matches any character not in *range*. The rules *R1–R4* set under 5) also apply here.

(7) *pattern*`?` stands for 0 or 1 copies of *pattern* where *pattern* is a specific character, the period `.` (meaning any character) or a range `[...]` as described under 5) and 6) or something in parentheses. *pattern*`*` stands for 0 or any number of concatenated copies of *pattern*. *pattern*`+` stands for 1 or any number of concatenated copies of *pattern*.

(8) The ordinary parentheses `(` and `)` are used for grouping.

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

A.2. Advanced Patterns in `awk`

Address patterns in `awk` that select the pattern space for action can be

(1) regular expressions as described in A.1,

(2) algebraic-computational expressions involving variables[20] 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.

Example: In the introduction, the following is used:

```
awk '(($1~/^between$/)||($(NF)~/^between$/))&&($0~/ through /)' -
```

This prints every line of input where the first or (`||`) last field equals `between` and (`&&`) there exits a field that equals `through` on the line by invoking the default action (*i.e.*, printing). It is assumed that fields are separated by blanks. This is used in the very first example code in the introduction.

[20] For example, the variable fields of the input record can be matched against patterns using the tilde operator.