Introduction to **awk** programming

(block course)

UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

Michael F. Herbst
michael.herbst@iwr.uni-heidelberg.de
http://blog.mfhs.eu

Interdisziplinäres Zentrum für wissenschaftliches Rechnen
Ruprecht-Karls-Universität Heidelberg

15th – 17th August 2016

Contents

Contents **i**

List of Tables **iv**

Course description **v**

Learning targets and objectives . . . v

Prerequisites . . . vi

Compatibility of the exercises **vi**

Errors and feedback **vi**

Licensing and redistribution **vi**

1 A first look at `awk` **1**

1.1 Design principles of `awk` . . . 1

1.2 `awk` versions and implementations . . . 2

1.3 `awk` programs . . . 2

1.3.1 Running `awk` programs . . . 2

1.4 Getting help and further reading . . . 5

2 Regular expressions **6**

2.1 Matching regular expressions in `awk` patterns . . . 6

2.2 Regular expression operators . . . 7

2.3 A shorthand syntax for bracket expansions . . . 10

2.4 POSIX character classes . . . 10

2.5 Getting help with regexes . . . 11

3 Basic features of `awk` **13**

3.1 Overview: How `awk` parses input . . . 13

3.2 Working with default `awk` input parsing . . . 14

3.3 Strings . . . 16

3.4 Multiple actions per pattern . . . 17

3.5 Variables . . . 18

3.5.1 Operators . . . 22

3.5.2 Arithmetic operators . . . 23

3.5.3 Conditional operators . . . 25

3.5.4 Conditional operators in patterns . . . 27

3.6 Standalone awk scripts 29
3.7 What we can do with awk so far 31

4 Influencing input parsing 33
4.1 Changing how files are split into records 33
4.2 Changing how records are split into fields 34
4.2.1 Using regular expressions to separate fields 35
4.2.2 Special field separator values 36
4.3 Defining fields by their content 38
4.4 Other ways to get input 39

5 Printing output 40
5.1 The print statement 40
5.1.1 Influencing the formatting of printed data 41
5.2 Fancier printing: printf 43
5.2.1 Format specifiers for printf 43
5.3 Redirection and piping from awk 47

6 Patterns, variables and control statements 49
6.1 Controlling program flow with rules and patterns 49
6.1.1 Range patterns 50
6.2 Control statements 52
6.2.1 exit statement 52
6.2.2 next statement 53
6.2.3 if-else statement 55
6.2.4 while statement 55
6.2.5 for statement 57
6.2.6 break statement 59
6.2.7 continue statement 60
6.3 Builtin and special variables in awk 60

7 Arrays 62
7.1 Statements and control structures for arrays 64
7.2 Multidimensional arrays 67

8 Functions 68
8.1 Important built-in functions 69
8.1.1 Numeric functions 69
8.1.2 String functions 70
8.2 User-defined functions 74

9 Writing practical awk programs 77
9.1 When and how to use awk 77
9.2 Re-coding Unix tools using awk 78
9.3 Example programs 79

A Obtaining the files 85

B Supplementary information 86
B.1 The mtx file format 86

Bibliography **88**

Index **89**

List of Tables

2.1 Some POSIX character classes 11

4.1 `awk`'s input parsing behaviour for different values of the field separator `FS`. 37

Course description

Dealing with large numbers of plain text files is quite frequent when making scientific calculations or simulations. For example, one wants to read a part of a file, do some processing on it and send the result off to another program for plotting. Often these tasks are very similar, but at the same time highly specific to the particular application or problem in mind, such that writing a single-use program in high-level language like `C++` or `Java` hardly ever makes much sense: The development time is just too high. On the other end of the scale are simple shell scripts. But with them sometimes even simple data manipulation becomes extremely complicated or the script simply does not scale up and takes forever to work on bigger data sets.

Data-driven languages like `awk` sit on a middle ground here: `awk` scripts are as easy to code as plain shell scripts, but are well-suited for processing textual data in all kinds of ways. One should note, however, that `awk` is not extremely general. Following the UNIX philosophy it can do only one thing, but this it can do right. To make proper use of `awk` one hence needs to consider it in the context of a UNIX-like operating system.

In the first part of the course we will thus start with revising some concepts, which are common to many UNIX programs and also prominent in `awk`, like *regular expressions*. Afterwards we will discuss the basic structure of `awk` scripts and core `awk` features like

- ways to define how data sits in the input file
- extracting and printing data
- control statements (`if`, `for`, `while`, …)
- `awk` functions
- `awk` arrays

This course is a subsidiary to the bash course which was offered in August 2015. See `http://blog.mfhs.eu/teaching/advanced-bash-scripting-2015/` for further information.

Learning targets and objectives

After the course you will be able to

- enumerate different ways to define the structure of an input file in `awk`,
- parse an structured input file and access individual values for post-processing,
- use regular expressions to search for text in a file,
- find and extract a well-defined part of a large file without relying on the exact position of this part,
- use `awk` to perform simple checks on text (like checking for repeated words) in less than 5 lines of code.

Prerequisites

- Familiarity with a UNIX-like operating system like GNU/Linux and the terminal is assumed.
- Basic knowledge of the UNIX command `grep` is assumed. You should for example know how to use `grep` to search for a word in a file.
- It is not assumed, but highly recommended, that participants have had previous experiences with programming or scripting in a UNIX-like operating system.

Compatibility of the exercises

All exercises and script samples have been tested on Debian 7 “Jessie” with GNU `gawk` 4.1.1. Other operating systems and `awk` implementations may not work. Most importantly your `awk` interpreter should understand the `gawk` dialect. See section 1.2 on page 2 and appendix A on page 85 for details.

Errors and feedback

If you spot an error or have any suggestions for the further improvement of the material, please do not hesitate to contact me under `michael.herbst@iwr.uni-heidelberg.de`.

Licensing and redistribution

Course Notes

An electronic version of this document is available from `http://blog.mfhs.eu/teaching/introduction-to-awk-programming-2016/`. If you use any part of my work, please include a reference to this URL along with my name and email address.

Script examples

Chapter 1

A first look at awk

Just like most other programming languages and computer programs, awk does not sit separate from the environment in which it is used. Much rather it is embedded into a larger context of other utilities and programs. Before we start discussing “proper” awk in chapter 3 on page 13 we will therefore quickly review the design principles of awk and how it is intertwined with other UNIX concepts.

1.1 Design principles of awk

awk was designed by Alfred Aho, Peter Weinberger, and Brian Kernighan in the 1970s at Bell Labs[1]. Their idea was to create a programming language for processing text files that was as easy to write as plain shell scripts, but which was more **data-driven**, i.e. where the structure of the data itself determines how the program is actually executed. As we will see in chapter 4 on page 33 this leads to a slightly unusual way of coding, but this will turn out to be very effective when reading files and extracting data. Often only a couple lines of code are enough to perform a rather intricate job[2].

Historically awk was influenced by shell scripting and the UNIX tools sed and grep, which can in fact be seen as a subset of awk. Right from the start it was distributed inside the UNIX operating system and even nowadays awk is a mandatory utility for every UNIX or Linux operating system[3].

As such it employs a lot of techniques which are very common to other UNIX tools as well. **Regular expressions** (see chapter 2 on page 6) for example are not only important to influence for awk's data parsing (see chapter 4 on page 33), but also play a prominent role in searching or editing text with grep, sed, vim, perl, python, …Furthermore most standard awk programs act more or less like a filter to the data which is passed through them. This implies that they can be easily used inside shell scripts in order to perform some processing which is required. We will only talk very briefly about this interplay of shell scripts (or other scripts) and awk in chapter 4 on page 33 and chapter 9 on page 77, but more details can be found in chapter 8 of the *Advanced bash scripting*

[1] That's the place where C, sh and a whole lot of other UNIX-related innovations comes from.

[2] Take a look at the examples at the end of the script (chapter 9 on page 77).

[3] See http://www.unix.org/version3/apis/cu.html and LSB 4.0

course [1].

1.2 `awk` versions and implementations

There exist three variants of the `awk` language:

- Old awk or `oawk`: The original version of the language by Aho, Weinberger and Kernighan from 1977.
- New awk or `nawk`: An extended `awk` language published in 1985.
- GNU awk or `gawk`: A yet extended version by the Free Software Foundation.

We will discuss only the dialect `gawk` here, since it is most common and has a couple of nice features the other versions lack.

Unfortunately by default most Linux distribution ship a program called `mawk` as their `awk` interpreter. So if some of the things explained in this script do not work as intended, one should make sure that the `gawk` program is installed.

1.3 `awk` programs

Similar to `sed` and `grep` an input file given to `awk` is automatically split up into larger chunks called **record**s. Usually this is done at the newline character, i.e. a record is in most cases equivalent to a line in the input file[4].

Each record is subsequently compared against a list of **pattern**s and if a pattern is satisfied a corresponding **action** is applied to the record. As such `awk` programs are listings similar to

```
pattern { action }
pattern { action }
...
```

Each line consisting of `pattern` and `action` is called a **rule**. It is important to note that for each record in a file *all* rules are executed. So the program itself contains an implicit loop over all records/lines of the input file.

As we will see in chapter 6 on page 49 quite some different types of `pattern`s and `action`s are supported.

1.3.1 Running `awk` programs

There are multiple ways to run `awk` programs. If the program is short it is easiest to include it on the commandline itself:

```
$ awk 'program' inputfile
```

This will read the file `inputfile` line-by-line and execute the `awk` code `program` on it.

[4] See section 4.1 on page 33 for cases where this is different

If programs become larger it is advantageous to store them in a file and read them from there. awk supports this using the flag -f. I.e.

```
$ awk -f programfile inputfile
```

will read the program code from the file programfile instead. awk scripts are usually given the extension awk.

Example 1.1. Consider the text files[5]

```
some words
any data
some further words
somer morer things
more other thing
even more data
```

resources/testfile

and

```
words
any data
more words
somer morer things
more other thing
even more data
```

1_first_look/smallerfile

- If we run the awk onelinear program

  ```
  /some/ { print }
  ```

  1_first_look/printo.awk

  i.e.

  ```
  $ awk -f 1_first_look/printo.awk resources/testfile
  ```

  we get the output

  ```
  some␣words
  some␣further␣words
  somer␣morer␣things
  ```

  since the pattern /some/ will only be satisfied if the line of text[6] under consideration contains the character sequence some. Of course one could have equally executed

  ```
  $ awk '/some/ { print }' resources/testfile
  ```

  to obtain the output above.

[5] All paths and resources of this script will be given relative to the top level directory of the git repository of this course (see appendix A on page 85). This is the directory which contains this very pdf file.

[6] Recall that by default a record is a line of text

- On the other hand we could run the same program on the input file `1_first_look/smallerfile`, i.e.

  ```
  $ awk -f 1_first_look/printo.awk 1_first_look/smallerfile
  ```

  to obtain

  ```
  somer␣morer␣things
  ```

  In this case only a single line matches the pattern `/some/`, so only a single line of output is printed.

Someone new to the concept of data-driven programming might find the output of the script `1_first_look/printo.awk` somewhat strange. Even though only a single line of code was written multiple lines of output were produced. As previously mentioned this is due to the implicit loop over records, i.e. the fact that *for each* record/line of the input file *all rules* of the `awk` program are executed. So with the input file `resources/testfile` where many lines satisfy the pattern, many lines are printed, but with the file `1_first_look/smallerfile` only one is printed.

Example 1.2. Printing the whole line is the default action of `awk` and the action block `{print}` may therefore be left out. Consider for example the output of

```
$ awk  '/more/' resources/testfile
```

where the action-block is missing. We get the output

```
somer␣morer␣things
more␣other␣thing
even␣more␣data
```

Exercise 1.3. We want to find all the lines of the Project Gutenberg[7] books `pg74` and `pg76`, which contain the word “hunger”.

- Write a simple `awk` programs to achieve your goal. If you proceeded with the setup of the repository as described in appendix A on page 85, you should find the books in the folder `resources/gutenberg`.
- *(optional)* Now write a small `awk` program file called `first.awk`, which finds lines which either contain the word “hunger” or “year”. Execute your script on `pg74` like this

  ```
  $ awk  -f first.awk resources/gutenberg/pg74.txt
  ```

Exercise 1.4. If a rule has no pattern, then all records match, i.e. the action is executed for all records. This explains why the commandline

```
$ awk '{print}' resources/testfile
```

prints the full content of `resources/testfile`, i.e.

```
some␣words
any␣data
```

[7] https://www.gutenberg.org/

```
some␣further␣words
somer␣morer␣things
more␣other␣thing
even␣more␣data
```

Now we want to understand the term **data-driven** programming language a little better.

- Explain the behaviour of

  ```
  $ awk -f 1_first_look/printprint.awk resources/testfile
  ```

  where the content of the `awk` script is

  ```
  {print}
  {print}
  ```

  1_first_look/printprint.awk

- What happens if the input file is empty? Create yourself an empty file using `touch empty` and execute

  ```
  $ awk -f 1_first_look/printprint.awk empty
  ```

  Why is this?

1.4 Getting help and further reading

For core utilities like `awk` there exist a vast number of potential sources for getting help. I personally very rarely use the internet, much rather I use the `gawk` manual: “GAWK: Effective AWK programming” [2]. It is in my opinion by far the best available `awk` resource. This course follows the structure of this book quite closely and for almost every aspect covered here further details can be found in [2]. I will refer to chapters and sections of this book for further reading at many places in this script.

For quickly checking an option or a some tiny detail, I can also recommend the `gawk` manpage, available from the terminal via the usual `man awk`.

Chapter 2

Regular expressions

Regular expressions are a key tool in order to find or describe strings in a text. As `awk` is all about processing text files we will need very often. For example they are used all the time to define the patterns, i.e. they play an important role to define when our actions are executed. The use of regular expressions is, however, not limited to `awk`. They are very important for a number of other UNIX utils like `grep`, `sed`, . . . as well. For more information about regular expressions in this latter context, see chapter 7 of [1].

2.1 Matching regular expressions in `awk` patterns

We will introduce regular expressions in a second, but in order to show some examples, we need a way to try them inside `awk` code. The `/word/` pattern we met in the previous chapter (see e.g. example 1.1 on page 3), is not just valid for searching for a string[1] in a record. Much rather it allows us to specify a regular expressions between the `/` characters. We will see later that regular expressions are just a generalisation to searching for plain string and therefore why we could use the `/regex/` pattern in order to search for strings in words as well.

Long story short: A simple `awk` program like

```
/regex/ {print}
```

already does the trick: It will check for each record/line of the input file whether at least a part of the record can be successfully described by the regular expression `regex` — one also says it checks whether at least a part of the line is **match**ed by the `regex`. If this is the case the action block `{print}` will be executed, i.e. the line will be printed to the output. Otherwise nothing happens for this line.

Example 2.1. The regex `r.t` matches all lines which contain an `r` and two characters later an `t`. So if we run

```
$ awk '/r.t/ {print}' resources/testfile
```

we get

[1] character sequence

```
somer␣morer␣things
more␣other␣thing
```

since the strings morer things and other thing are matched by `r.t`. Of course a word like rotten would have been matched as well. It is important to note here that really the full record/line is matched irrespective of the individual words.

To quickly check whether a string `string` is matched by a `regex` it is pretty tedious to first put the `string` into an `inputfile` and then execute

```
$ awk '/regex/ {print}' inputfile
```

Luckily `awk` reads from the standard input if no `inputfile` is given, i.e. the commandline

```
$ echo "string" | awk '/regex/ {print}'
```

can also be used[2] Of course we could also drop the action block `{print}` entirely since it is the default action executed anyway (see example 1.2 on page 4):

```
$ echo "string" | awk '/regex/'
```

2.2 Regular expression operators

It is best to think of regular expressions as a “search” string where some characters have a special meaning. All non-special characters just stand for themselves, e.g. the regex “`a`” just matches the string “`a`”[3].

Without further ado a non-exhaustive list of **regular expression operators**[4]:

`\` The escape character: Disables the special meaning of the next character that follows.

`^` matches the beginning of a string, ie. “`^word`” matches “`wordblub`” but not “`blubword`”. It is important to note for later, that `^` does not match the beginning of a line, but much rather the beginning of a record.

```
$ echo "wordblubb" | awk '/^word/'
```

gives

```
wordblubb
```

[2] Recall that a **pipe** “`|`” can be used to redirect the output of one program to be the input of another. In this case `echo` prints the string as its output, which is redirected to be the input of `awk`. For more details about this see section 2.3 of [1].

[3] This is why in example 1.1 on page 3 we could just use the `/regex/` pattern to search for strings in the text without really knowing anything about regexes.

[4] More can be found e.g. in section 3.3 of the `awk` manual [2]

whilst

```
$ echo "blubbword" | awk '/^word/'
```

yields no output.

`$` matches the end of a string or record in a similar way.

`.` matches any single character, including `<newline>`, e.g. `P.P` matches `PAP` or `PLP` but not `PLLP`

`[...]` **bracket expansion**: Matches one of the characters enclosed in square brackets.

```
$ echo "o" | awk '/^[oale]$/'   # match
```

```
o
```

```
$ echo "a" | awk '/^[oale]$/'   # match
```

```
a
```

```
$ echo "oo" | awk '/^[oale]$/'  # no match
```

Note: Inside the bracket expansion only the characters `]`, `-` and `^` are *not* interpreted as literals. E.g.

```
$ echo '$' | awk '/^[$]$/'      # match
```

```
$
```

`[^...]` **complemented bracket expansion**: Matches all characters *except* the ones in square brackets

```
$ echo "o" | awk '/[^eulr]/'    # match
```

```
o
```

```
$ echo "e" | awk '/[^eulr]/'    # no match
```

```
$ echo "a" | awk '/[o^ale]/'    # match, since this is no cbe!
```

```
a
```

`|` **alternation operator**: Specifies alternatives: Either the regex to the right or the one to the left has to match. Note: Alternation applies to the largest possible regexes on either side

```
$ echo "word" | awk '/^wo|rrd$/'   # match, since ^wo
```

```
word
```

`(...)` Grouping regular expressions, often used in combination with `|`, to make the alternation clear, e.g.

```
$ echo "word" | awk '/^(wo|rrd)$/  # no match'
```

```
```

`*` The *preceding* regular expression should be repeated as many times as necessary to find a match, e.g. “`ico*`’ matches “`ic`”, “`ico`” or “`icooooo`”, but not “`icco`”. The “`*`” applies to the *smallest* possible expression only.

```
echo "wo␣(rd" | awk '/wo* \(/'         # match
echo "woo␣(rd" | awk '/wo* \(/'        # match
echo "oo␣(rd" | awk '/wo* \(/'         # no match
echo "oo␣(rd" | awk '/(wo)* \(/'       # match
echo "wowo␣(rd" | awk '/(wo)* \(/'     # match
```

2_regular_expressions/regex_star.sh

```
wo␣(rd
woo␣(rd
oo␣(rd
wowo␣(rd
```

`+` Similar to “`*`”: The preceding expression must occur at least once

```
echo "woo␣(rd" | awk '/wo+ \(/'     # matches
echo "oo␣(rd" | awk '/(wo)+ \(/'    # no match
echo "wo␣(rd" | awk '/(wo)+ \(/'    # matches
```

2_regular_expressions/regex_plus.sh

```
woo␣(rd
wo␣(rd
```

`?` Similar to “`*`”: The preceding expression must be matched once or not at all. E.g. “`ca?r`” matches “`car`” or “`cr`”, but nothing else.

There are a few things to note

- `awk` will try to match as much of a record as possible.
- Regexes are case-sensitive
- Unless `^` or `$` are specified, the matched substring may start and end anywhere.
- As soon as a single matching substring exists in the record, the record is considered to match the pattern, i.e. the action will be executed for this record.

2.3 A shorthand syntax for bracket expansions

Both bracket expansion and complemented bracket expansion allow for a shorthand syntax, which can be used for *ranges* of characters or ranges of numbers, e.g

short form	equivalent long form
`[a-e]`	`[abcde]`
`[aA-F]`	`[aABCDEF]`
`[^a-z4-9A-G]`	`[^abcdefgh ... xyz456789ABCDEFG]`

Exercise 2.2. Consider these strings

“ab” “67” “7b7”
“g” “67777” “o7x7g7”
“77777” “7777” “” (empty)

For each of the following regexes, decide which of the above strings are matched:

- `..`
- `^..$`
- `[a-e]`
- `^.7*$`
- `^(.7)*$`

2.4 POSIX character classes

There are also some special, named bracket expansions, called **POSIX character classes**. See table 2.1 on the following page for some examples and [2] for more details. POSIX character classes can only be used within bracket expansions, e.g.

```
# ^[[:space:]]*[0[:alpha:]]+ matches arbitrarily many spaces
# followed by at least one 0 or letter

echo "␣␣␣␣␣a" | awk '/^[[:space:]]*[0[:alpha:]]+/'  # Match
echo "␣00" | awk '/^[[:space:]]*[0[:alpha:]]+/'     # Match
echo "1" | awk '/^[[:space:]]*[0[:alpha:]]+/'       # No match
echo "␣␣␣1" | awk '/^[[:space:]]*[0[:alpha:]]+/'    # No match
```

2_regular_expressions/regex_posixclass.sh

which gives the output

```
␣␣␣␣␣a
␣00
```

short form	equivalent long form	description
`[:alnum:]`	`a-zA-Z0-9`	alphanumeric chars
`[:alpha:]`	`A-Za-z`	alphabetic chars
`[:blank:]`	`␣\t`	space and tab
`[:digit:]`	`0-9`	digits
`[:print:]`		printable characters
`[:punct:]`		punctuation chars
`[:space:]`	`␣\t\r\n\v\f`	space characters
`[:upper:]`	`A-Z`	uppercase chars
`[:xdigit:]`	`a-fA-F0-9`	hexadecimal digits

Table 2.1: Some POSIX character classes

2.5 Getting help with regexes

Writing regular expressions takes certainly a little practice, but is extremely powerful once mastered.

- `https://www.debuggex.com` is extremely helpful in analysing and understanding regular expressions. The website graphically analyses a regex and tells you why a string does/does not match.
- Practice is everything: See `http://regexcrossword.com/` or try the Android app *ReGeX*.

Exercise 2.3. Fill the following regex crossword. The strings you fill in have to match both the pattern in their row as well as the pattern in their column.

	`a?[3[:space:]]+b?`	`b[^21eaf0]`
`[a-f][0-3]`		
`[[:xdigit:]]b+`		

Exercise 2.4. Give regular expressions that satisfy the following

	matches	does not match	chars
a)	`abbbc`, `abbc`, `abc`, `ac`	`aba`	2
b)	`abbbc`, `abbc`, `abc`	`bac`, `ab`	3
c)	`ac`, `abashc`, `a123c`	`cbluba`, `aefg`	2
d)	`␣␣qome`, `␣qol` , `qde`	`eqo`, `efeq`	3
e)	`arrp`, `whee`	`bla`, `kee`	4

Note: The art of writing regular expressions is to use the smallest number of characters possible to achieve your goal. The number in the last column gives the number of characters necessary to achieve *a* possible solution.

Exercise 2.5. *(optional)* Take a look at the file `resources/digitfile`. This file contains both lines which contain only text as well as lines which contain numbers. The aim of this exercise is to design a regular expression which matches only those lines that contain numbers in proper scientific format, i.e. a number of the form

$$\underline{\texttt{sign}}\,\underline{\texttt{prefactor}}\,\texttt{e}\,\underline{\texttt{sign}}\,\underline{\texttt{exponent}}$$

e.g.

```
0.123e-4    0.45e1    -0.4e9
```

These numbers follow the rules

- The `sign` may be `+` or `-` or absent
- The `prefactor` has to be in the range between `0.` and `1.` In other words it will always contain a `.` as the second digit and the first character will always be a 0 or 1. The number of characters after the `.` may, however, vary.
- The `exponent` may be any integer number, i.e. it may not contain a `.`, but otherwise any number. It may have leading zeros.

In order to design the regular expression, proceed as follows:

- First design regexes to match the individual parts: `sign`, `prefactor` and `exponent`.
- Paste them together. Pay attention to which parts are required and which are optional.
- *(optional)* Introduce some fault tolerance:
  - Make your expression work if between prefactor and exponent one of the characters `E`, `D` or `d` is used instead.
  - Be less strict on the requirements of the prefactor. Allow prefactors outside of the range `0` to `1`.

Chapter 3

Basic features of awk

Now that we understand regular expressions we return our attention to `awk` itself. Our ultimate goal when using `awk` is to parse data and work on it in some way. Whilst we will look more closely into influencing the parsing process itself in the next chapter 4 on page 33, this chapter will discuss the basics of `awk`. We will look at core features like dealing with strings, variables and operators.

3.1 Overview: How awk parses input

Consider once again the basic structure of an `awk` program with a list of rules consisting of patterns and actions:

```
pattern { action }
pattern { action }
...
```

For any data `awk` gets to parse — either in form of an input file that we specify on the commandline or piped on standard input — the following is underdone:

1. The whole input data is split into smaller chunks, called **record**s. This done whenever a **record separator** character is encountered. By default the record separator is the `<newline>` character, which is sometimes also denoted as `\n`.
2. Each record is further split into **fields** whenever a **field separator** is found in the record. By default newline and all members from the `[:space:]` POSIX character class (see table 2.1 on page 11) are field separators.
3. For each record all rules are considered from top to bottom. If the pattern of a rule matches for a given record, then the corresponding action is executed.
4. If the action block is absent, the whole record will be printed (*default action*).
5. If the pattern is absent (*default pattern*), the action is executed for each word.

In other words default `awk` input parsing will go through the file line by line (records) and will assign to the individual fields the words[1] it encounters in each line.

[1] In the sense that words are separated by `[:space:]` characters.

Of course the number of words in a line of input may differ, which demonstrates that the number of fields available for a record does not need to be a constant number.

3.2 Working with default `awk` input parsing

Before any rule is executed for a particular record, the special variables `$1`, `$2`, . . . , `$n` are assigned to the first, second, . . . n-th field of this record. For example consider a record like

```
Das Pferd frisst keinen Gurkensalat.
```

`Das` will be assigned to `$1`, `Pferd` will be assigned to `$2` and so on. These variables can be easily used in an action block:

```
echo "Das␣Pferd␣frisst␣keinen␣Gurkensalat." | awk '{print $2}'
```

yields

```
Pferd
```

The whole record as it was read from the input is stored in the variable `$0` as well, such that

```
echo "Das␣Pferd␣frisst␣keinen␣Gurkensalat." | awk '/^Das/ {print $0}'
```

gives us

```
Das␣Pferd␣frisst␣keinen␣Gurkensalat.
```

since the line starts with “`Das`”.

Exercise 3.1. Use an `awk` oneliner to extract the *second* word of each line of the book `resources/gutenberg/pg1232.txt`, which starts with the string `free`.

Variables like the field variables are not static to `awk`. This means that we can change them to a new value, e.g. a string like `"Auto"` (More on strings in section 3.3 on page 16):

```
$ echo "Das␣Pferd␣frisst␣keinen␣Gurkensalat." | awk '{$2 = "Auto"}
                                                     {print}'
```

```
Das␣Auto␣frisst␣keinen␣Gurkensalat.
```

We notice that `awk` takes changes to variables into account when the next rule is processed (in this case a simple unconditional `{print}` action).

Example 3.2. Suppose we manage a list of telephone numbers like

```
Amelia⎵555-5553⎵1
Anthony⎵555-3412⎵2
Becky⎵555-7685⎵1
Bill⎵555-1675⎵4
Broderick⎵555-0542⎵5
Camilla⎵555-2912⎵2
Fabius⎵555-1234⎵0
Julie⎵555-6699⎵3
Martin⎵555-6480⎵1
Samuel⎵555-3430⎵2
Jean-Paul⎵555-2127⎵3
```

resources/data/telephonelist

(For clarity we use the symbol ⎵ to denote a tab and ␣ to denote a space character in these kinds of outputs.)

Our job is to change the telephone number of all people starting with the letter "`B`" to "`not_available`". Since `awk` takes changes to the variables `$1`, `$2` ... into account when processing all rules below the current one, this can be achieved using the `awk` program

```
# Match lines starting with B
# and change the second field
/^B/ { $2 = "not_available" }
# Print all records including the changed one
{ print }
```

3_basics/change_avail.awk

Running this on `resources/data/telephonelist`, i.e.

```
$ awk -f 3_basics/change_avail.awk resources/data/telephonelist
```

results in

```
Amelia⎵555-5553⎵1
Anthony⎵555-3412⎵2
Becky␣not_available␣1
Bill␣not_available␣4
Broderick␣not_available␣5
Camilla⎵555-2912⎵2
Fabius⎵555-1234⎵0
Julie⎵555-6699⎵3
Martin⎵555-6480⎵1
Samuel⎵555-3430⎵2
Jean-Paul⎵555-2127⎵3
```

We note that the formatting of the output lines differs when printing unaltered records and records where `awk` was asked to change the value of a field. This can be explained as follows:

- `awk` removes repeated whitespace characters when splitting the record into fields, no matter what the whitespace is (tabs, spaces, newlines, ...), so the field variables `$1`, `$2`, etc. only contain the plain words and no whitespace

- The unchanged record (exactly as it was read from input) is retained in `$0`.
- If the fields are not changed, then `print $0` hence prints the original record as read.
- If one field is changed, `$0` is rebuilt from the individual fields. By default this means the field values are concatenated together, separated by only a *single* space. We will see in section 5.1.1 on page 41 how one can alter this behaviour.

Notice, that all the lines starting in the program code of the previous example, which started with `#` were ignored by `awk`. This is because `#` is the **comment character in** `awk`. Everything which is placed behind a `#` on the same line will be ignored.

3.3 Strings

Strings in `awk` all have to be enclosed by double quotes, e.g.

```
"This␣is␣a␣valid␣string"
```

Multiple strings may be concatenated, just by leaving white space between them

```
"This␣is␣a␣"      "␣concatenated␣"      "string"
```

This way we can concatenate the content of variables and some other text before printing:

```
$ echo "Das␣Pferd␣frisst␣keinen␣Gurkensalat" | awk '{ print "Der␣" $5 "␣" $3 "␣kein␣" $2 }'
```

```
Der␣Gurkensalat␣frisst␣kein␣Pferd
```

Note that we had to include explicit spaces between `$5` and `$3` because the fields have leading and trailing whitespace removed.

A nice feature of `awk` is that it per default honours special escape sequences like "`\t`"(for a tab character) and "`\n`"(for a newline character) in strings. So we could improve the output formatting in example 3.2 on the previous page by explicitly putting two tabs between the fields, i.e.

```
# Match lines starting with B
# and change the second field
/^B/ { $2 = "not_available" }
# Print all records separated by two tabs
{ print $1 "\t\t" $2 "\t\t" $3 }
```

3_basics/change_avail_tabs.awk

Running this on `resources/data/telephonelist` gives

```
Amelia␣␣555-5553␣␣1
Anthony␣␣555-3412␣␣2
Becky␣␣not_available␣␣1
Bill␣␣not_available␣␣4
Broderick␣␣not_available␣␣5
Camilla␣␣555-2912␣␣2
```

```
Fabius␣␣␣␣555-1234␣␣␣␣0
Julie␣␣␣555-6699␣␣␣3
Martin␣␣␣␣555-6480␣␣␣␣1
Samuel␣␣␣␣555-3430␣␣␣␣2
Jean-Paul␣␣␣555-2127␣␣␣␣3
```

which is still not great, but certainly nicer to read. We will pick up on the output formatting issue in chapter 5 on page 40.

Exercise 3.3. Use an awk oneliner to extract the first and the second column from the file resources/matrices/lund_b.mtx.
(optional) Modify your code to prevent data from the comment lines (starting with the character %) to be printed. This is easiest if you use a regular expression in the pattern.

3.4 Multiple actions per pattern

As awk programs become more complicated it is often desired to execute more than one action per matching pattern. For example consider the program

```
# Match lines starting with B
# inform us about the unavailable phone and change the field.
/^B/ { print $1 "'s␣phone␣has␣become␣unavailable"
  $2 = "not_available" }
# Print all records separated by two tabs
{ print $1 "\t\t" $2 "\t\t" $3 }
```

3_basics/change_avail_message.awk

here we not only want to change the field if the phone of a person has become unavailable, but we also want to print a statement about this to the screen. This can be achieved by putting the two actions

```
print $1 "'s␣phone␣has␣become␣unavailable"
```

and

```
$2 = "not_available"
```

into different lines of code. The resulting output is

```
Amelia␣␣␣␣555-5553␣␣␣␣1
Anthony␣␣␣555-3412␣␣␣␣2
Becky's␣phone␣has␣become␣unavailable
Becky␣␣␣not_available␣␣␣1
Bill's␣phone␣has␣become␣unavailable
Bill␣␣␣␣not_available␣␣␣4
Broderick's␣phone␣has␣become␣unavailable
Broderick␣␣␣not_available␣␣␣5
Camilla␣␣␣555-2912␣␣␣␣2
Fabius␣␣␣␣555-1234␣␣␣␣0
Julie␣␣␣555-6699␣␣␣3
Martin␣␣␣␣555-6480␣␣␣␣1
Samuel␣␣␣␣555-3430␣␣␣␣2
Jean-Paul␣␣␣555-2127␣␣␣␣3
```

Since empty lines in action blocks or empty lines between different rules are ignored by awk we could equivalently write the program code as

```
# Match lines starting with B
# inform us about the unavailable phone and change the field.
/^B/ {
  print $1 "'s␣phone␣has␣become␣unavailable"
  $2 = "not_available"
}

# Print all records separated by two tabs
{ print $1 "\t\t" $2 "\t\t" $3 }
```

3_basics/change_avail_message2.awk

which is easier to read.

awk also allows to replace line breaks between actions or rules by ; (semicolon) characters. This is especially helpful for small commandline awk programs, e.g.

```
$ echo "awk␣is␣twice␣not␣cool." | awk '{$4="as"
                                       print}
                                      {print}'
```

and

```
$ echo "awk␣is␣twice␣not␣cool." | awk '{$4="as";print};{print}'
```

both produce

```
awk␣is␣twice␣as␣cool.
awk␣is␣twice␣as␣cool.
```

3.5 Variables

The handling of variables and arithmetic in awk are very similar to other scripting languages like bash or python. Most things should be pretty self-explanatory and this section is therefore kept rather short. More details about variables and operators in awk can be found in section 6.1 of the gawk manual [2].

- Variables are assigned using a single equals ("="). There may well be space between the name and the value.

```
var="value"
# or
var =    "value"
# are identical to awk
```

- The value of a variable can just be accessed by its name:

```
print "var␣is␣" var       # => will print "var is value"
```

- Variables do not need to hold string values:

```
integer=1       # an integer
float=3.4       # a floating point number
float2=1.3e7    # another floating point number
```

- awk performs sensible conversion between strings that describe numbers and other variables/strings

```
{
  var1 = "1.23"
  var2 = 1.1

  print "With␣$1=" $1 ":␣" $1 + var1
  print var1 "+" var2 "=" var1+var2
}
```

3_basics/vars_conversion.awk

```
$ echo "111" | awk -f 3_basics/vars_conversion.awk
```

```
With␣$1=111:␣112.23
1.23+1.1=2.33
```

- Strings that cannot be interpreted as a number are considered to be 0.

```
$ echo "blubber" | awk -f 3_basics/vars_conversion.awk
```

```
With␣$1=blubber:␣1.23
1.23+1.1=2.33
```

- Undefined variables are 0 or the empty string.
- All variables are global and can be accessed and modified from all action blocks (or condition statements as we will see later)

```
# $0 is always defined to be the current record,
# but A is undefined at this point
{
  print "$0␣is␣" $0 "␣but␣A␣is␣" A
  N = 15
}

# print N and define A
{ print N; A = $1 }

# print A
{ print "String␣" A "␣has␣length␣" length(A) }
```

3_basics/vars_global.awk

```
$ echo "blubber" | awk -f 3_basics/vars_global.awk
```

```
$0␣is␣blubber␣but␣A␣is
15
String␣blubber␣has␣length␣7
```

Note that the state of `awk` is not reset between processing different records. In other words a variable, which is set once, can be used when processing a different record as well[2] This can be a little confusing at first, but is actually one of the reasons, why `awk` is so effective in processing text files.

Example 3.4. To make this point more clear, let us consider an example:

```
{
  print "current:␣" $1 "␣" $2
  print prev "␣with␣buffer␣" buffer
}

# set the buffer if there is an i in the name
# and the 12 occurs on the line as well (e.g. in the phone number)
/i.*12/ { buffer = $2 }

{
  prev = $1 "␣" $2     # store the current record in prev
  print ""             # print an empty line
}
```

3_basics/vars_rule_global.awk

```
$ awk -f 3_basics/vars_rule_global.awk resources/data/telephonelist
```

```
current:␣Amelia␣555-5553
␣with␣buffer

current:␣Anthony␣555-3412
Amelia␣555-5553␣with␣buffer

current:␣Becky␣555-7685
Anthony␣555-3412␣with␣buffer

current:␣Bill␣555-1675
Becky␣555-7685␣with␣buffer

current:␣Broderick␣555-0542
Bill␣555-1675␣with␣buffer

current:␣Camilla␣555-2912
Broderick␣555-0542␣with␣buffer

current:␣Fabius␣555-1234
```

[2] In some sense `awk` is pretty much a state machine, where parsing records can induce transitions between different internal states.

```
Camilla␣555-2912␣with␣buffer␣555-2912

current:␣Julie␣555-6699
Fabius␣555-1234␣with␣buffer␣555-1234

current:␣Martin␣555-6480
Julie␣555-6699␣with␣buffer␣555-1234

current:␣Samuel␣555-3430
Martin␣555-6480␣with␣buffer␣555-1234

current:␣Jean-Paul␣555-2127
Samuel␣555-3430␣with␣buffer␣555-1234
```

- The program runs over our list of phone numbers. It stores some phone number in a buffer, called `buffer`. Furthermore line 12 achieves that the first and the second field of the current record is stored inside the variable `prev`. Then processing ends with printing an empty line.
- The behaviour of line 3 of the `awk` code differs when processing the first record compared to the remaining ones.

  For the *first record* the variable `prev` is not yet defined. For *all others* this line causes the first two fields from the previous line to be printed. This happens because `prev` has just been set to this value when the *previous* record was processed.
- In other words line 12 of the program influences the behaviour in line 3, i.e.

  Code at the end can have an effect on code at the beginning!

  Whether such a bottom-to-top influence really happens, depends on the input data (data-driven programming) and what rules match for each of the records.

⇒ When writing an `awk` program one does not only need to think sequentially from top to bottom, but sometimes the other way round as well. E.g. when a one needs to cache a part of a record for consumption when processing a later record.

Exercise 3.5. Write an `awk` program, which prints an ever-growing concatenation of all previous lines. In other words the input data

```
awk
is
a
data-driven
programming
language
```

3_basics/input.data

should give rise to

```
␣awk
␣awk␣is
␣awk␣is␣a
␣awk␣is␣a␣data-driven
␣awk␣is␣a␣data-driven␣programming
```

```
␣awk␣is␣a␣data-driven␣programming␣language
```

The extra space in the beginning is a little annoying, but we do not know a way to avoid it at the moment.

Exercise 3.6. Explain why the script

```
{ print a }
{ num = "false"; a = a + 1 }
/[0-9]/ { num="true"; a = a - 1 }
{ print "num:␣" num }
```

3_basics/exscript.awk

yields the output

```

num:␣true
0
num:␣true
0
num:␣false
1
num:␣false
2
num:␣true
2
num:␣true
2
num:␣true
```

3_basics/exscript.awk.out

when executed with the input file

```
4
5
word
no␣more
3␣little
wild␣5animals
11
```

3_basics/exscript.awk.in

3.5.1 Operators

awk allows all sorts of operations between variables and objects. Some of these operations (like the string concatenation we mentioned in section 3.3 on page 16 do not even require an explicit operator to be present). Amongst the operators we distinguish between two major categories — arithmetic operators (section 3.5.2 on the following page), which act solely on numbers, and conditional operators (section 3.5.3 on page 25), which compare objects of various kinds.

3.5.2 Arithmetic operators

`awk` supports the following arithmetic operators[3]:

`x ^ y` Exponentiation: `x` is raised to the power of `y`

`-x` Negation

`+x` Unary plus: `x` is converted to a number

`x * y` Multiplication

`x / y` Division: Since `awk` is floating-point aware the result is *not always* an integer. E.g. `3/4` will be `0.75`. If you want to enforce integer division, use the expression `int(3/4)` which will yield `0` in this case.

`x % y` Modulo operation or remainder of integer division

`x + y` Addition

`x - y` Subtraction

For example

```awk
{
  # calculate the square and print it
  e = $1 ^ 2
  print e

  # Assign a string variable:
  str="3.44444444444444444444"
  # convert to a number and print it
  print +str
  # print it without conversion
  print str

  # print sum
  print str+$2

  # print integer and normal division
  print int(5/3), 5/3

  # show that operator precedence makes sense
  print 5+3*2.5
}
```

3_basics/arith_example.awk

```
$ echo "12.3␣4" | awk -f 3_basics/arith_example.awk
```

```
151.29
3.44444
3.44444444444444444444
7.44444
1␣1.66667
12.5
```

[3] See chapter 6.2 of [2] for more details

Similar to the C-like languages, there exist combined arithmetic assignment operators like +=, -= or similar. Increment ++ and decrement -- are also supported in the usual manor.

```
# Count number of lines with a as second character
/^.a/ { acount+= 1 }

# Count the number of lines containing 2
/2/  { twocount++ }

{
  print "So␣far␣found␣" acount "␣'a's␣as␣second␣character"
  print "So␣far␣found␣" twocount "␣lines␣containing␣'2'"
  print "--"
}
```

3_basics/arith_incr_example.awk

```
$ awk -f 3_basics/arith_incr_example.awk resources/data/telephonelist
```

```
So␣far␣found␣␣'a's␣as␣second␣character
So␣far␣found␣␣lines␣containing␣'2'
--
So␣far␣found␣␣'a's␣as␣second␣character
So␣far␣found␣1␣lines␣containing␣'2'
--
So␣far␣found␣␣'a's␣as␣second␣character
So␣far␣found␣1␣lines␣containing␣'2'
--
So␣far␣found␣␣'a's␣as␣second␣character
So␣far␣found␣1␣lines␣containing␣'2'
--
So␣far␣found␣␣'a's␣as␣second␣character
So␣far␣found␣2␣lines␣containing␣'2'
--
So␣far␣found␣1␣'a's␣as␣second␣character
So␣far␣found␣3␣lines␣containing␣'2'
--
So␣far␣found␣2␣'a's␣as␣second␣character
So␣far␣found␣4␣lines␣containing␣'2'
--
So␣far␣found␣2␣'a's␣as␣second␣character
So␣far␣found␣4␣lines␣containing␣'2'
--
So␣far␣found␣3␣'a's␣as␣second␣character
So␣far␣found␣4␣lines␣containing␣'2'
--
So␣far␣found␣4␣'a's␣as␣second␣character
So␣far␣found␣5␣lines␣containing␣'2'
--
So␣far␣found␣4␣'a's␣as␣second␣character
So␣far␣found␣6␣lines␣containing␣'2'
--
```

Of cause it is very annoying to get all the intermediate results if one is only interested in the final count. For this reason the special pattern END exists, which is only matched exactly once: After the processing of all records has been done. So if we use the program

```
# Count number of lines with a as second character
/^.a/ { acount+= 1 }

# Count the number of lines containing 2
/2/  { twocount++ }

END {
  print "We␣found␣" acount "␣'a's␣as␣second␣character"
  print "We␣found␣" twocount "␣lines␣containing␣'2'"
}
```

3_basics/arith_incr_example_end.awk

on resources/data/telephonelist we just get

```
We␣found␣4␣'a's␣as␣second␣character
We␣found␣6␣lines␣containing␣'2'
```

which much better. A similar BEGIN pattern, which matches before any record is read, exists as well. It is most often used to initialise variables.

Exercise 3.7.

- Use awk to count how many lines of resources/digitfile contain some number.
- Use the END pattern to only print a final result.
- *(optional)* If you did exercise 2.5 on page 11: Find out how many lines contain actual scientific numbers.

Exercise 3.8. Use awk to find the average value of the first, second and third columns in resources/matrices/3.mtx. In other words compute one average for the first, one for the second and one for the third column. Run your code on matrices/lund_b.mtx and matrices/bcsstm01.mtx.

3.5.3 Conditional operators

- awk has the usual comparison operators[4]

x == y	True if x and y are equal
x != y	True if x and y are not equal
x < y	True if x is smaller than y
x > y	True if x is greater than y
x <= y	True if x is smaller or equal to y
x >= y	True if x is greater or equal to y

[4] See chapter 6.3 of [2] for more details

- The kind of comparison performed by these operators is determined by the types of `x` and `y`. The precise rules are a bit involved and rarely of importance. They can be found in section 6.3.2 of the [2]. A couple of examples:

```
1.5 <= 2.0      # numeric comparison (true)
"abc" >= "xyz"  # string comparison (false)
1.5 != "␣+2"    # string comparison (true)
"1e2" < "3"     # string comparison (true)
a=2; b="2"
a==b            # string comparison (true)
a=2; b="+2"
a==b      # string comparison (false)
```

  A good rule of thumb is that string comparison is performed unless the strings originate from user input, i.e. are stored in field variables like `$1`, `$2`, . . .

- Similar to the `C`-like languages, "true" is equivalent to 1 and "false" is equivalent to 0, e.g.

```
$ echo "1␣2" | awk '{ print ($1 < $2); print ($1 > $2) }'
```

```
1
0
```

- `awk` further implements two special **regex comparison operators**, namely

  `x ~ y` True if `x` matches the regex denoted by `y`

  `x !~ y` True if `x` does not match `y`

  In both of this cases `y` may either be a regex literal like `/regex/` as well as a variable or expression which makes up a regular expression. For example

```
1  {
2    var="Some␣words␣in␣a␣variable."
3
4    # use a regex literal
5    print "var␣matches␣/variable\\./␣" (var ~ /variable\./)
6
7    # Use a regex variable
8    re="le\\.$"
9    print "var␣matches␣re?␣" (var ~ re)
10   # Note: In order to precisely match the "." at the end of
11   # var we need to escape the ".", i.e. use "\.".
12   # This however would be interpreted by awk when making the
13   # string variable re in line 8 before we reach the regex
14   # in line 9.  So we need to escape the escape first ...
15   # Since this is not necessary for the regex literals
16   # (see line 5) Those are usually preferred.
17
18   # use an expression to build the regex
19   build="-9]"
20   print "var␣does␣not␣match␣[0" build "?␣" (var !~ "[0" build)
21
22   # does $0 match?
```

```
  print (var ~ $0)
}
```

3_basics/cond_regex_example.awk

```
$ echo "ds..n" | awk -f 3_basics/cond_regex_example.awk
```

```
var␣matches␣/variable\./␣1
var␣matches␣re?␣1
var␣does␣not␣match␣[0-9]?␣1
1
```

- Finally it is possible to logically connect conditions using `||` (logical or), `&&` (logical and) or `!` (logical not), e.g.

```
echo "1" | awk '{print ($0 || (3 > 4)) && !("a" > "z")}'
```

```
1
```

```
echo "0" | awk '{print ($0 || (3 > 4)) && !("a" > "z")}'
```

```
0
```

3.5.4 Conditional operators in patterns

- All conditional operators we met in section 3.5.3 on page 25 may be used in the pattern section of a rule. For example

```
$ echo "blub␣ber" | awk '$2 ~ /be/ { print "matches" }'
```

```
matches
```

or

```
$ echo "awk␣course" | awk '$2 == "course" { print ↙
    ↪"blubber!!" }'
```

```
blubber!!
```

- Using the Boolean operators `&&` or `||` many conditions may be may well be checked in the pattern together.

```
$ echo "awk␣course" | awk '/^a.. / && $2 ~ /r.e/ { print ↙
    ↪"more␣blubber!!" }'
```

```
more␣blubber!!
```

- As we saw above this way we can explicitly check for the values of variables before executing an action.

Example 3.9. Suppose we want to print all entries of `resources/data/telephonelist` which have a third column of 2 or higher. We want to do the printing in chunks of two, such that after each two findings there is an empty line. The `awk` program

```
# if 3rd column is larger than 2 print it
# and increment counter c
$3 >= 2 { print; c++ }

# if we did some printing and the print count
# is divisible by 2 add an extra empty line
$3 >= 2 && c % 2 == 0 { print ""}
```

3_basics/chunk_2.awk

does exactly that:

```
Anthony␣555-3412␣␣2
Bill␣␣555-1675␣␣4

Broderick␣555-0542␣␣5
Camilla␣555-2912␣␣2

Julie␣555-6699␣␣3
Samuel␣␣555-3430␣␣2

Jean-Paul␣555-2127␣␣3
```

Exercise 3.10. This exercise deals once again with the matrix files `resources/matrices/3.mtx`, `resources/matrices/bcsstm01.mtx` and `resources/matrices/lund_b.mtx`. These `mtx` files are representing matrices stored in the matrix market format. Take a look at appendix B.1 on page 86 for some details about this format.

- Write an `awk` program that prints both the number of expected explictly provided entries as well as the number of actual entries found in the last indices-value block of the `mtx` file.
- Write an `awk` program that computes the sparsity ratio of the three matrix files. The sparsity ratio of an $n \times m$ matrix is defined as

$$1 - \frac{\text{Number of entries, which are stored}}{n \cdot m}.$$

In the case of such an `mtx` file the "Number of entries, which are stored" is equivalent to the number of explictly provided entries.
(optional) Modify your program such that it does not rely on the correctness of the value d in the first non-comment line.

- *(optional)* Write `awk` code, which computes the elementwise square of a `mtx` file. Your output should be a valid `mtx` file as well, so be careful not to operate on the first non-comment line!

3.6 Standalone awk scripts

Up to now we know a couple of different ways to run awk programs. We can pipe some input to the awk executable and provide the program code on the commandline, like

```
$ echo "2␣3" | awk '{ print $1 + $2 }'
```

```
5
```

Or we can write our code to a file and let awk read it from there

```
{ print $1+$2 }
```

3_basics/add.awk

```
$ echo "2␣3" | awk -f 3_basics/add.awk
```

```
5
```

In both those cases we may also provide the input data as a file, e.g.

```
$ awk -f 3_basics/add.awk resources/matrices/3.mtx
```

```
0
6
2
3
4
3
4
5
4
5
6
```

There exists yet another option, namely writing **awk scripts**. Similar to other shell scripts[5] we need to do two things:

- Provide the **shebang**

  ```
  #! /usr/bin/awk -f
  ```

  on the very first line of our program, in order to tell the operating system that the following code should be executed with awk.
- Make the script file executable by calling chmod +x on it.

In our case we need to modify 3_basics/add.awk to

```
#!/usr/bin/awk -f
{ print $1+$2 }
```

3_basics/add_script.awk

[5] See section 3.1 of [1]

and then execute

```
$ chmod +x 3_basics/add_script.awk
```

After this has been achieved, we can use it as a regular script. For example we can pipe data to it

```
$ echo "2␣3" | 3_basics/add.awk
```

```
5
```

or call it with an input file as first argument

```
$ 3_basics/add.awk resources/matrices/3.mtx
```

```
0
6
2
3
4
3
4
5
4
5
6
```

The advantage is that this script can be used like any other program of the operating system once it is placed into the `$HOME/bin` folder. See chapter 3 of [1] for details.

3.7 What we can do with awk so far

Now we have covered enough `awk` to achieve the most basic tasks. A couple of examples:

- Print lines longer than 80 characters:

```
#!/usr/bin/awk -f
length($0) > 80
```

3_basics/ex_longer.awk

- Print the length of the longest line

```
#!/usr/bin/awk -f
length($0) > max { max = length($0) }
END { print "The␣longest␣line␣has␣" max "␣characters." }
```

3_basics/ex_longest_line.awk

- Discard duplicated lines of input:

```
#!/usr/bin/awk -f
$0 != prev
{ prev = $0 }
```

3_basics/ex_unique.awk

- Print everything from the line containing `heading` until an empty line is reached

```
#!/usr/bin/awk -f
# if line is empty, quit
$0 == "" { exit }

# as soon as we hit a line called heading set pr=1
/heading/ { pr=1 }
# print if pr equals 1
pr == 1
```

3_basics/ex_headingpart.awk

Here we used the action command `exit` which instructs `awk` to immediately quit. See section 6.2.1 on page 52 for details.

Exercise 3.11. *(demo)* The file `resources/chem_output/qchem.out` contains the logged output of a quantum-chemical calculation. A common problem when performing scientific calculation is to extract the relevant data from such an output. This exercise will deal with writing an `awk` script which aids with extraction of important information of so-called *excited states*.

- If one wants to achieve such a task with `awk`, one usually first tries to find a suitable character sequence, which surrounds our regions of interest. We will then use a state variable like `pr` in the example program `3_basics/ex_headingpart.awk` to switch our main processing routine on and off[6].

[6] `awk` has better ways to do this. See section 6.1.1 on page 50.

- Take a look at lines 565 to 784 of `qchem.out`. In this case we are interested in creating a list of the 10 computed excited states. For each state the list should contain the state's number, its term symbol (e.g. "`1 (1) A"`" or "`3 (1) A'`") and its excitation energy.
- For the processing of the first state we hence need only the six lines

```
␣␣Excited␣state␣1␣(singlet,␣A")␣␣␣␣␣␣␣␣␣␣␣␣[converged]
␣␣----------------------------------------------------
␣␣Term␣symbol:␣␣1␣(1)␣A"␣␣␣␣␣␣␣␣␣␣␣␣R^2␣=␣␣7.77227e-11

␣␣Total␣energy:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-7502.1159223236␣a.u.
␣␣Excitation␣energy:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣3.612484␣eV
```

  to extract the information

  - Term symbol: `1 (1) A"`
  - State number: 1
  - Excitation energy `3.612484 eV`

  Similarly for the other excited states blocks.

Now proceed to write the script:

- Decide for a good starting and a good ending sequence.
- How you would extract the data (state number, term symbol, excitation energy) once `awk` parses the excited states block?
- Be careful when you extract the term symbol, because the data will sit in more than one field.
- Cache the extracted data for an excited states block until you reach the ending sequence. Then print it all at once in a nicely formatted table.

Chapter 4

Influencing input parsing

`awk`'s default input parsing (see section 3.1 on page 13) turns out to be very good for most things. Still it is desirable to tell `awk` as precisely as possible in which format the input data is to be expected. This way as much of the parsing work as possible is already done in the background by `awk` and we can concentrate on working with the actual data instead.

This chapter only gives a rough overview what we can do to influence the input parsing. Some features and many subtleties are not covered here. See chapter 4 of [2] for more details.

4.1 Changing how files are split into records

When `awk` reads input it splits it into records at each occurrence of the **record separator**, by default the `<newline>` character. This behaviour can be changed by altering the built-in variable `RS`.

Usually one uses the `BEGIN` special pattern (see 6.1 on page 49) change `RS`, because the action block corresponding to the `BEGIN` pattern will be executed *before* any input is read. For example[1]

```
$ echo -n "And␣he␣shouted:␣'Ooh␣ya␣koo␣␣!'" | awk 'BEGIN { ↙
   ↪RS="o"}; {print $0}'
```

produces

```
And␣he␣sh
uted:␣'O
h␣ya␣k

␣␣!'
```

- Since the `RS="o"` changes the record separator to "`o`" a new record is started each time the letter `o` is encountered.

[1] `echo -n` is used in order to suppress the extra `<newline>` `echo` usually prints

- The second rule causes each record to be printed. Note that `print` adds a `<newline>` after the string `$0`, regardless of the value of `RS`.
- If the record separator occurs multiple times just after itself, an empty record is yielded.

Field splitting is not influenced by the altered value of `RS`[2], e.g. we get

```
$ echo -n "And␣he␣shouted:␣'Ooh␣ya␣koo␣␣!'" | awk 'BEGIN { ↙
    ↪RS="o"}; {print $1 "--" $2 "--" $3}'
\end{lstinline}
\begin{lstlisting}[style=output]
And--he--sh
uted:--'O--
h--ya--k
----
!'----
```

where as usual repeated whitespace separates the fields.

Exercise 4.1. Write an `awk` program which finds duplicated words in a text and prints the duplicated words it found, but produces no other output.
Hint: This is pretty much the same problem as removing duplicated lines in a file if you alter `RS` properly.

`awk` does not only allow single characters as record separators, but also e.g. regular expressions and many more. More details about this can be found in 4.8 of [2].

4.2 Changing how records are split into fields

The analogue to the special variable `RS` for influencing field splitting is the **field separator** `FS`. Whenever `awk` encounters the value of `FS`, a new field begins. Typical non-default values for `FS` are "`,`", "`;`" or "`:`". Consider for example

```
$ echo "one::1:␣:" | awk 'BEGIN { FS=":" }; { print $1 "--" $2 ↙
    ↪"--" $3 "--" $4 "--" $5 }'
```

```
one----1--␣--
```

We notice:

- Altering the field separator implies that whitespace is no longer stripped off when determining the fields content.
- Repeated occurrence of the field separator gives rise to empty fields (like `$3` in this case).

Exercise 4.2. A very common class of files are so-called **comma-separated value files**. Many online-banking websites and almost all spreadsheet programs allow to export their data as such files. Typically the individual data fields are separated by the same character, usually a "`,`" (comma).

[2] This is not fully correct. See 4.8 of [2] for details.

Consider the file `resources/data/money.csv`. This file contains a statement with a bunch of transfers. The initial balance is given in the first line starting with a `#`. The following lines (line 4 and onwards) contain the actual transfers, given by a description and the respective change in balance.

- Write an `awk` program which prints the final balance to the screen.
- Write an `awk` program which appends a third column which gives the current balance after each transfer has occurred.

Sometimes it is desirable to change the value of `FS` in the middle of an `awk` program. This has, however, no effect on the current fields. Only the way subsequent records are separated into fields is changed. E.g.

```
$ awk '/^B/ { FS="-" }; { print $1 }' resources/data/telephonelist
```

yields

```
Amelia
Anthony
Becky
Bill	555
Broderick	555
Camilla	555
Fabius	555
Julie	555
Martin	555
Samuel	555
Jean
```

Once the first record starting with `B` is read, `FS` is changed. But this only affects the parsing of the second record starting with `B` and all subsequent records.

4.2.1 Using regular expressions to separate fields

A slightly more general way to split records into fields is to use a regular expression for `FS`. Any assignment to `FS` which is more than one character long is interpreted as a regular expression by `awk`. For example the assignment `FS=",\t"` causes “comma followed by tab“ to be the new field separator:

```
$ echo "one,	two, and three,	four" | awk 'BEGIN { FS=",\t" }; { print $1 "--" $2 "--" $3 "--" $4  }'
```

```
one--two, and three--four--
```

It is, however, more common to use regexes involving a repeated bracket expansion like `FS="[:,]+"` for the field separator. In this case this would start a new field whenever one ore more “:” or “,” are encountered.

Example 4.3. A very important file for a UNIX operating system is `/etc/passwd`. This file contains information about all users of the system and uses a combination of "`:`" and "`,`" to separate fields. The script

```
#!/usr/bin/awk -f

# set field separator to be : or , or many of these chars
BEGIN { FS="[:,]+" }

# give a nice listing of the current entry
# for all uids >= 1000
$3 >= 1000 {
  print $1
  print "␣␣␣uid:␣␣␣␣␣␣␣" $3
  print "␣␣␣full␣name:␣" $5
  print "␣␣␣home␣dir:␣␣" $6
  print "␣␣␣shell:␣␣␣␣␣" $7
}
```

4_parsing_input/passwd_summary.awk

prints a nice summary for the users with user id greater or equal than 1000.

4.2.2 Special field separator values

In section 4.2 on page 34 we discussed how `FS` could be changed to single character in order to make this character the field separator. Whenever this character occurred a repeated number of times in the record this resulted in empty fields.

This is in contrast to the behaviour, which one finds if `FS` is set to `"␣"`. This latter case is the default value of `FS` in `awk` and triggers that records are split whenever a single or repeated occurrence of *any* whitespace character is encountered. So it actually behaves more or less as if `FS` was assigned to `[ \t\n]+`[3].

Example 4.4. Contrast the output of

```
$ echo "one::1:␣3:" | awk 'BEGIN { FS=":" }; { print $1 "--" $2 "--" $3 "--" $4 "--" $5 }'
```

```
one----1--␣3--
```

with

```
$ echo "one⎵⎵1⎵␣3⎵" | awk 'BEGIN { FS="␣" }; { print $1 "--" $2 "--" $3 "--" $4 "--" $5 }'
```

where all `:` have been replaced by tab in the `echo` command. The result is this time

```
one--1--3----
```

since all fields but `$1`, `$2` and `$3` are empty.

[3] But unfortunately not exactly so . . . see 4.5.2 of [2] for details.

`FS=" "`	(default) Fields are separated by single or repeated occurrences of any whitespace character. Leading or trailing whitespace is ignored.
`FS="c"`	where `c` is any single character. Fields are separated at each occurrence of `c`. Multiple successive occurrences give rise to empty fields. Trailing occurrences are *not* ignored and also lead to empty fields.
`FS="regex"`	Fields are separated by occurrences of characters or character sequences which match the `regex`. Any string value with more than one character, which is assigned to `FS` is interpreted as a regex.
`FS=""`	Each individual character in the record becomes a separate field.

Table 4.1: `awk`'s input parsing behaviour for different values of the field separator `FS`.

If one wants to enforce that field splitting occurs exactly at space characters and nowhere else and that repeated spaces trigger empty fields, one needs to set `FS` to the regex “`[ ]`”. E.g.

```
$ echo "one␣␣1␣␣3␣" | awk 'BEGIN { FS="[␣]" }; { print $1 "--" ↙
    ↪$2 "--" $3 "--" $4 "--" $5 }'
```

```
one----1----3
```

We see that now `$1` is empty, whereas `$2` is “1” and `$5` is “3”.

Another special value for `FS` is `""` (empty string). In this case each character is a field on its own.

```
#!/usr/bin/awk -f
BEGIN { FS="" }
{
  print "F1->" $1 "<-"
  print "F2->" $2 "<-"
  print "F3->" $3 "<-"
}
```

4_parsing_input/ex_each_char_field.awk

```
$ echo "a␣b" | 4_parsing_input/ex_each_char_field.awk
```

```
F1->a<-
F2->␣<-
F3->b<-
```

A summary of the possible values for `FS` and how this alters the input parsing process is summarised in table 4.1.

4.3 Defining fields by their content

So far our approach to describe the input data was to provide `awk` with values for the variables `FS` and `RS`. Both variables have in common that we specify the separator characters — either directly or by the means of a regular expression. In other words we describe what our data *is not* rather than to describe what our data actually *is*.

In principle specifying `RS` and `FS` has the advantage that one does not need to know so well what the values of the fields in our data look like. This makes the resulting programs very general and applicable for many files of a given type.

Sometimes it is, however, considerably easier to describe what kind of data sits in a field rather than describing what separates the fields. E.g. consider the `csv` file

```
#dialect,year,authors
awk,1977,"Alfred Aho, Peter Weinberger, Brian Kernighan"
nawk,1985,"Brian Kernighan"
mawk,1992,"Mike Brennan"
gawk,1988,"Paul Rubin, Jay Fenlason, Richard Stallman"
```

resources/data/awk.csv

which contains three columns. The third column is a string which may itself contain whitespace or commas. As a consequence the naive approach

```
$ awk 'BEGIN {FS=","}; { print $1 "," $3 "," $2 } resources/data/awk.csv
```

does not get us very far[4].

A solution to this problem offers the `FPAT` variable. Its value should be a regular expression, which describes the contents of *each* field. It is **greedy** in order to determine the value of each field, which means that it tries to match the regular expressions with as much data as possible. Consequently one needs to make sure that the regex is not too broad, because otherwise the whole record is a single field.

In the case of our `csv` file a good pattern to describe each field's value is `([^,]+)|("[^"]+")`, i.e. either anything but a comma or anything but double quotes. When we assign this value to `FPAT`, we need to be careful to properly escape the double quotes (since these also delimit strings in `awk`). Overall this yields the program

```
#!/usr/bin/awk -f
BEGIN { FPAT="([^,]+)|(\"[^\"]+\")" }
{ print $1 "," $3 "," $2 }
```

4_parsing_input/ex_csv_swap.awk

which applied to `resources/data/awk.csv` gives:

```
#dialect,authors,year
awk,"Alfred␣Aho,␣Peter␣Weinberger,␣Brian␣Kernighan",1977
nawk,"Brian␣Kernighan",1985
mawk,"Mike␣Brennan",1992
gawk,"Paul␣Rubin,␣Jay␣Fenlason,␣Richard␣Stallman",1988
```

[4] This might give you already a hint what a horrible and totally underdefined file format `csv` is . . .

Exercise 4.5. *(optional)* Use your regular expression pattern from 2.5 on page 11 to add all scientific numbers in `resources/digitfile` and print the result.
If you want you can also try to add all numbers (regardless whether they are integers, floats or scientific numbers).

4.4 Other ways to get input

So far we covered how one may influence the way `awk` deals with input data, which was provided on standard input or by passing a file upon execution.

`awk` is, however, much more flexible. It may, for example, read an arbitrary number of files at a time or dynamically execute a command and interpret its output. The relevant command is called `getline` and it is very powerful. For further information on using `getline` see section 4.9 of the `awk` manual [2].

Chapter 5

Printing output

The ultimate goal of every awk program is to output some data which was extracted or computed from the parsed input. For this purpose we already used the print statement in the previous chapters.

In this chapter we will both discuss ways to affect the formatting of the printed results as well as introduce the printf command which allows much more fine-grained specification of how things are formatted. Finally in section 5.3 on page 47 we will briefly touch on awk's capabilities for writing data to files or pipe to other commands.

5.1 The print statement

print allows to pass the value of one or many variables to standard output. E.g.

```
print item1, item2, ...
```

will first print the value of item1 followed by a space and the content of item2 and so on. After everything has been printed a <newline> character is added. If more than one item is specified it is recommended (but not required) to include the arguments in parenthesis, i.e.

```
print(item1, item2, ...)
```

Example 5.1. Compare the output of

```
$ awk '{ print $1, $2 }' resources/data/telephonelist
```

```
Amelia␣555-5553
Anthony␣555-3412
Becky␣555-7685
Bill␣555-1675
Broderick␣555-0542
Camilla␣555-2912
Fabius␣555-1234
Julie␣555-6699
Martin␣555-6480
```

```
Samuel␣555-3430
Jean-Paul␣555-2127
```

to

```
$ awk '{ print $1 $2 }' resources/data/telephonelist
```

```
Amelia555-5553
Anthony555-3412
Becky555-7685
Bill555-1675
Broderick555-0542
Camilla555-2912
Fabius555-1234
Julie555-6699
Martin555-6480
Samuel555-3430
Jean-Paul555-2127
```

In the first case the comma makes sure that `print` gets passed the two arguments `$1` and `$2` separately, which are as a consequence printed with a space in between. In the second case the strings `$1` and `$2` are first concatenated and then passed as one argument to `print`. Hence no space shows up.

`awk` honours special characters in printed string literals. This allows to print more than one line in a single `print` command or to include tabs in the printed output. For example

```
$ awk '{ print $1 "\thas␣the␣number:\n" $2 "\n" }' ↙
    ↪resources/data/telephonelist
```

```
Amelia␣␣has␣the␣number:
555-5553

Anthony␣has␣the␣number:
555-3412

Becky␣has␣the␣number:
555-7685

Bill␣␣has␣the␣number:
555-1675
...
```

5.1.1 Influencing the formatting of `printed` data

Both the separator between items as well as the separator at the end of the printed data are configurable:

- The special variable `OFS` (output field separator) controls how the individual printed items are separated (default: `OFS="␣"`).

- `ORS` (output record separator) controls which character is printed at the end of a `print` (default: `ORS="\n"`, i.e. `<newline>`)

We mentioned in section 3.2 on page 14 that altering a field variable like `$1`, `$2`, . . . causes `$0` to be rebuilt. This uses the value of `OFS` in order to separate fields. E.g.

```
$ echo "a␣b␣c" | awk 'BEGIN {OFS=":"}; {$2 = "2"; print $0}'
```

```
a:2:c
```

Contrast this to

```
$ echo "a␣b␣c" | awk 'BEGIN {OFS=":"}; {print $0}'
```

```
a␣b␣c
```

where no rebuild has occurred (no field value has been altered) and hence `$0` still contains the original record. If one wants to trigger a rebuild without changing any field value, a very common practice is to use a dummy assignment like `$1=$1`:

```
$ echo "a␣b␣c" | awk 'BEGIN {OFS=":"}; {$1=$1; print $0}'
```

```
a:b:c
```

Example 5.2. Many important environment variables like `PATH` or `LD_LIBRARY_PATH` hold a colon-separated string of directories, e.g. a typical value of `PATH` is

```
/usr/local/bin:/usr/bin:/bin
```

Often the configuration of these variables, however, is realised in files which just contain one directory per line. A simple `awk` program like

```
#!/usr/bin/awk -f
# print records separated by :
BEGIN { ORS=":" }
{ print }
```

5_printing_output/fold_folders.awk

can read the input files and produce the value for `PATH`.

Exercise 5.3. Write `awk` programs in order to achieve the following in less than 3 lines of code.

- Unfold a text such that each word will be printed to its own line. There are two solutions. Try your programs on some of the resource files, e.g. `resources/testfile`.
- Change the separator character in `resources/data/money.csv` from comma(`,`) to semicolon(`;`).

5.2 Fancier printing: printf

For more control over the formatting of the printed data awk provides the command printf (*print fancy*)[1] Its basic syntax is

```
printf(format, item1, item2, ...)
```

The main difference to print is the extra format string argument in first position. This argument defines how awk interprets the other argument items and how their value is formatted. Most characters of the format string are not treated specially, but are printed verbosely to the output instead. For example

```
$ echo "10␣1.2␣3␣4" | awk '{printf("Data:␣\nand␣more")}'
```

```
Data:
and␣more
```

Note, that printf does neither automatically append a newline nor the value of ORS to the output[2] If one wants the line to end after a printf command, one needs to manually insert a line break by adding a “\n” at the end of the format string.

5.2.1 Format specifiers for printf

A **format specifier** is a string starting with % and ending with a **format control letter**. This letter defines what *kind* of value to print (e.g. integer, string or floating point number). There also exist so-called **format modifiers** which control *how to* print this value (e.g. how many significant figures to print).

In order to define the formatting of the individual items, printf expects *one* format specifier *for each* item.

The most important format control letters are[3]

%d, %i Print a decimal integer (%d and %i are absolutely equivalent)

```
$ echo "10␣1.2␣3␣4" | awk '{printf("Numbers:␣%i␣%i␣↙
    ↪%d\n",$1,$2,$3/$4)}'
```

```
Numbers:␣10␣1␣0
```

%e, %E Print a number in scientific notation, e.g.

```
$ echo "10␣1.2␣3␣4" | awk '{printf("Numbers:␣%e␣↙
    ↪%4.3e\n",$1,$3/$4)}'
```

```
Numbers:␣1.000000e+01␣7.500e-01
```

[1] Those which know the printf function of the programming language C will notice that awk’s printf is more or less exactly the same thing.

[2] In fact the behaviour of the printf statement is entirely unaffected by changes to the values of ORS or OFS.

[3] The full list can be found in section 5.5.2 of [2].

Here `4.3` is a format modifier, which will be discussed below. `%E` uses “E” instead of “e” in the output.

`%f` Print a number in fixed point notation, e.g.

```
$ echo "10 1.2 3 4" | awk '{printf("Numbers: %4.3f %4.3f %f\n",$1,$2,$3/$4)}'
```

```
Numbers: 10.000 1.200 0.750000
```

`%g, %G` Print a number in either scientific notation or fixed point notation, whichever uses fewer characters. If scientific notation is employed `%G` uses uppercase “E” instead of “e”. E.g.

```
$ echo "10 1.2 3 4e9" | awk '{printf("Numbers: %4.3g %4.3g %g\n",$1,$2,$3/$4)}'
```

```
Numbers:   10  1.2 7.5e-10
```

`%s` Print as a plain string (as if `print` was used on this item).

```
$ echo "10 1.2 3 4" | awk '{printf("Numbers: %s %s %s\n",$1,$2,$3/$4)}'
```

```
Numbers: 10 1.2 0.75
```

`%%` Print a literal `%`.

The most important format modifiers are[4]

`width` This is a number specifying the minimum width of a field. If one inserts `width` between the `%` and the format control letter the field is expanded to this width by padding the value with spaces on the left. E.g.

```
$ echo "foo" | awk '{printf(":%5s:\n", $1)}'
```

```
:  foo:
```

Note that `width` specifies the *minimum* width, i.e. fields may still expand beyond this width:

```
$ echo "long" | awk '{printf(":%5s:\n", $1 "word")}'
```

```
:longword:
```

`.prec` `prec` is a number specifying the precision of the numerical value to be printed. Its meaning depends on the control letter:

`%d,%i`	Minimum number of digits to print
`%e,%E,%f`	Number of digits to the right of the decimal point.

[4] The full list can be found in section 5.5.3 of [2].

`%g,%G` Maximum number of significant digits.

`%s` Maximum number of characters from the string to be printed.

For example

```
$ echo "long" | awk '{printf(":%.5s:\n", $1 "word")}'
```

```
:longw:
```

`-` If a minus sign is used before specifying the width modifier, the padding will be applied on the right, i.e. the data will be *left-justified*

```
$ echo "foo" | awk '{printf(":%-5s:\n", $1)}'
```

```
:foo␣␣:
```

`0` A leading zero before the width indicates than numerical values should be padded with zero, e.g.

```
$ echo "15" | awk '{printf(":%04d:\n", $1)}'
```

```
:0015:
```

Since the format string argument of `printf` is just a plain string it may well be built by string concatenation, e.g.

```
#!/usr/bin/awk -f

{
  width=10
  prec=4

  # build format string
  # the resulting string is "3 args: %10.4e %10.4f"
  format="3␣args:␣"
  format=format "%" width "." prec "e␣%" width "." prec "f"

  printf(format, $1/$2, $2-$3)
}
```

5_printing_output/format_vars.awk

```
$ echo "3␣15␣-1e-3" | 5_printing_output/format_vars.awk
```

```
3␣args:␣2.0000e-01␣␣␣␣15.0010
```

This is especially advantageous if `awk` programs become more complex and the precise number of fields is dynamic and only known at run-time.

Example 5.4. In this example we want to produce a nicely formatted table containing the information of `resources/data/telephonelist`. One way of doing this in a fairly flexible manor would be

```
#!/usr/bin/awk -f
BEGIN {
  # For the first column (name, allow 12 characters
  # - enforces justification to the left)
  name_width = 12
  format = format "%-" name_width "s"

  # Column separator
  format = format "␣|␣"

  # For the second column (phone number) allow
  # 8 characters, right-justify
  phone_width = 8
  format = format "%" phone_width "s"

  # Column separator
  format = format "␣|␣"

  # For the number of years employed allow 8
  # characters
  years_width = 8
  format = format "%" years_width "s\n"

  # Print the headline
  printf(format,"Name","Phone-No","yrs␣empld")
}
{ printf(format,$1,$2,$3) }
```

5_printing_output/format_telephone.awk

Running

```
$ 5_printing_output/format_telephone.awk resources/data/telephonelist
```

gives

```
Name␣␣␣␣␣␣␣␣␣|␣Phone-No␣|␣yrs␣empld
Amelia␣␣␣␣␣␣␣|␣555-5553␣|␣␣␣␣␣␣␣␣1
Anthony␣␣␣␣␣␣|␣555-3412␣|␣␣␣␣␣␣␣␣2
Becky␣␣␣␣␣␣␣␣|␣555-7685␣|␣␣␣␣␣␣␣␣1
Bill␣␣␣␣␣␣␣␣␣|␣555-1675␣|␣␣␣␣␣␣␣␣4
Broderick␣␣␣␣|␣555-0542␣|␣␣␣␣␣␣␣␣5
Camilla␣␣␣␣␣␣|␣555-2912␣|␣␣␣␣␣␣␣␣2
Fabius␣␣␣␣␣␣␣|␣555-1234␣|␣␣␣␣␣␣␣␣0
Julie␣␣␣␣␣␣␣␣|␣555-6699␣|␣␣␣␣␣␣␣␣3
Martin␣␣␣␣␣␣␣|␣555-6480␣|␣␣␣␣␣␣␣␣1
Samuel␣␣␣␣␣␣␣|␣555-3430␣|␣␣␣␣␣␣␣␣2
Jean-Paul␣␣␣␣|␣555-2127␣|␣␣␣␣␣␣␣␣3
```

Exercise 5.5. The file `resources/data/values.csv` contains the values of the same measurements made on 14 different apparatus. The first column contains the apparatus number and the remaining columns (separated by `:`) contain the measured values. The first two lines (starting with `#`) are comments.

- For each apparatus compute the average measurement value. Also compute the total average value in the same `awk` program.
- Show your results in a nice table similar to the example above. Give 3 significant figures for all values.
- *(optional)* One instrument seems to be a bit off. Which one? Exclude it when calculating the total average.

5.3 Redirection and piping from `awk`

Similar to plain shell scripts `awk` allows to redirect any output to a file or the standard input of a shell command. We do not want to go into details here, but just show a couple of examples demonstrating the feature. Details can be found in chapter 5.6 of [2].

Example 5.6. In this example we want to split our telephonelist into a file containing only the names and a file containing only the phone numbers. One way to do this is

```
#!/usr/bin/awk -f

BEGIN {
  # define name of name file
  namefile="name.list"

  # define name of phone file
  phonefile="phone.list"
}

{
  # Append name to namefile (all existing
  # content in namefile will be kept)
  print $1 >> namefile

  # Write phone number to phonefile
  # The single > indicates that this is no append
  # i.e. that all content of the file will be erased
  # before awk touches it. All subsequent writes
  # will still append to the file
  print $2 > phonefile
}
```

5_printing_output/split_telephone.awk

Example 5.7. Suppose we wrote a program `send_fax` which sends a fax to a phone number. If we want to execute this for everyone in the telephonelist, we could do this using awk like so

```
#!/usr/bin/awk -f

{
  # $1 contains the name
  # $2 contains the phone number

  # We want to pipe some text to the program, which
  # we want to fax to the person
  text="Hello␣" $1 "\n"
  text=text "I␣will␣call␣you␣soon␣for␣further␣info.\n"
  text=text "Best"

  # We call send_fax with the number as argument
  print text | ("send_fax␣" $2)

  # This is one of the subtleties when using this feature
  # with awk: Sometimes a manual close() is necessary.
  close("send_fax␣" $2)
}
```

5_printing_output/fax_list.awk

Chapter 6

Patterns, special variables and control statements

One of the key aspects of learning any programming language is knowing ways how to influence which parts of the program are executed when. In awk this is mainly realised by specifying a pattern in front of the actions: The actions are done exactly when the record matches the pattern. Furthermore the usual control structures like loops, if statements, etc. exist in awk as well. We will introduce them in section 6.2 on page 52.

Already at this point a word of caution: Control structures introduce a new level of flexibility to awk code. On the one hand this means that they allow to solve previously unsolvable problems. On the other hand control structures somewhat compete with awk-internal features — as we will see further down in this chapter. Additionally programs with too many control structures tend to violate the data-driven design paradigms behind awk: No longer the data, but some built-in program logic decides what to do.

6.1 Controlling program flow with rules and patterns

Before we deal with control structures, just a reminder and a summary about rules and patterns. We already saw before that an awk program is essentially made up of a list of pattern-action rules. Once the program runs each input record is compared against the patterns and in case the record matches a pattern the corresponding action block is executed.

This implies that in a usual awk program the patterns control which part of the program are executed and hence how execution flows between the action blocks. In other words: The patterns are the core feature in awk to control the **program flow**.

A summary of the type of patterns which are supported by awk:

/regex/ If the text of current record matches this regex the pattern is considered a match. See section 3.1 on page 13 for details.

`expr` A single expression, which usually involves comparison of some variables. In case the final result is non-zero or a non-empty string the pattern is considered a match. See section 3.5.4 on page 27 for details.

`BEGIN` Special pattern, which matches right before any input record is processed. Usually used for initialising variables.

`END` Special pattern, which matches right before the `awk` program is exited, either because there are no more records to process or because the `exit` statement has been encountered (see section 6.2.1 on page 52). Usually used to specify cleanup actions or print final results.

`pat1, pat2` A pair of patterns, which identifies a range of records which match. The range is inclusive on both sides, i.e. the first record is the one which matches `pat1` and the last record is the one which matches `pat2`. This type of pattern will be discussed in more detail in section 6.1.1 below.

empty The empty pattern matches all records.

As we saw in section 3.5.4 on page 27 two patterns may be logically combined using `||` (or) or `&&` (and) in order to form a more complex pattern for a rule.

Example 6.1. This example prints every second record, which contains a vowel.

```
#!/usr/bin/awk -f

/[aeiou]/ {
  # invert flag
  flag = !flag
}

# Print if condition and flag
# are satisfied.
/[aeiou]/ && flag
```

6_patterns_actions_variables/ex_vowel.awk

If we run this on `resources/data/telephonelist` we get

```
Amelia␣␣555-5553␣␣1
Becky␣555-7685␣␣1
Broderick␣555-0542␣␣5
Fabius␣␣555-1234␣␣0
Martin␣␣555-6480␣␣1
Jean-Paul␣555-2127␣␣3
```

6.1.1 Range patterns

Range patterns consist of two elementary patterns separated by “,”. For example the `awk` program

```
$1 == "on", $1 == "off"
```

will simply print every record between `on` and `off` pairs, including the delimiting `on/off` records itself. Quite often one employs regexes as the subpatterns to mark beginning and end.

Example 6.2. All Project Gutenberg books contain some information about the project itself as well as some licensing information, both of which are not part of the actual book. So if we want to do some processing on the book itself we need to make sure to explicitly exclude the paragraphs Project Gutenberg added.

Careful inspection of the Project Gutenberg files reveals that there are guarding lines like

```
*** START OF THIS PROJECT GUTENBERG EBOOK ... ***
```

and

```
*** END OF THIS PROJECT GUTENBERG EBOOK ... ***
```

where “...” stands for the book title in capitals in both cases. Hence the `awk` program

```
#!/usr/bin/awk
/^\*\*\* START OF THIS PROJECT GUTENBERG EBOOK/, /\*\*\* END OF THIS PROJECT GUTENBERG EBOOK/ { print tolower($0) }
```

6_patterns_actions_variables/pg_filter.awk

performs both the task of filtering out the non-interesting parts of the file and normalising all the strings to lower case (a typical first step when computer-processing text).

There are a couple of rather subtle points one needs to keep in mind when working with range patterns:

- If a record matches both beginning and end at the same time, then the action is executed *only for this record*. In other words we cannot use a pattern like

  ```
  /<-->/,/<-->/ { some_actions }
  ```

  to match a range between two occurrences of `<-->`. Instead we need something like

  ```
  # swap flag whenever <--> encountered
  /<-->/ { skip = ! skip }
  skip == 1 { some_actions }
  ```

- Range patterns cannot be combined with other patterns, i.e. constructs like

  ```
  $1 == "on", $1 == "off" && $2 == "blubber" { some_actions }
  ```

  do not work as expected. See section 7.1.3 of [2] for details.

Exercise 6.3. Write an `awk` program which extracts a given chapter from the book `resources/gutenberg/pg161.txt`. Your program should expect the chapter number to be stored in the variable `v`. Set `v` to various values in the BEGIN section in order to verify that your code works. Do not worry if you extract an extra line at the end or so.

Exercise 6.4. *(optional)* The file `resources/chem_output/qchem.out` contains the logged output of a quantum-chemical calculation. During this calculation two so-called Davidson diagonalisations have been performed. Say we wanted to extract how many iterations steps were necessary to finish these diagonalisations. Take a look at line 422 of this file. You should notice:

- Each Davidson iteration start is logged with the line

```
␣␣Starting␣Davidson␣...
```

- A nice table is printed afterwards with the iteration count given in the first column
- The procedure is concluded with the lines

```
------------------------------------------------------
␣␣Davidson␣Summary:
```

Use what we discussed so far about `awk` in order to extract the number of iterations both Davidson diagonalisations took.

6.2 Control statements

Similar to other programming languages `awk` offers a range of control structures like `for` or `while` loops and `if` statements to control the program flow[1]. These statements are, however, only allowed *inside action blocks*.

As mentioned before it should be avoided to overuse control statements in `awk` and instead rely on other `awk`-specific paradigms like patterns or a careful selection of the field separator. In my experience this makes best use of `awk`'s powers and as a result `awk` code become both shorter and less complicated.

Those of you familiar with a `C`-like programming language will probably notice that the `awk` control structures are designed with `C` syntax in mind.

6.2.1 `exit` statement

The `exit` statement instructs `awk` to quit the program. The current action block is left immediately and no further input is processed. If an `END` rule exists it is still executed, then the program is stopped. The full syntax of the statement is

```
exit return_code
```

The `return_code` argument is an optional integer between 0 and 126, which serves as the exit code of the `awk` program. By convention `0` means "success". Just `exit` is equivalent to `exit 0`. For example

```
#!/usr/bin/awk -f
{ print }
/^B/ { exit 15 }
END { print "Reached␣END" }
```

6_patterns_actions_variables/ex_exit.awk

```
$ 6_patterns_actions_variables/ex_exit.awk resources/data/telephonelist; echo $?
```

[1] `awk` has a lot more control structures than we discuss. See section 7.4 of [2] for the full details.

```
Amelia␣␣555-5553␣␣1
Anthony␣555-3412␣␣2
Becky␣555-7685␣␣1
Reached␣END
15
```

whereas

```
$ 6_patterns_actions_variables/ex_exit.awk resources/integers; echo $?
```

```
40
400
50000000000000
40000000000000
-10
Reached␣END
0
```

Especially the return-code feature is extremely helpful when combining `awk` programs with `bash` scripts[2].

6.2.2 next statement

The `next` statement causes `awk` to stop processing the current record and proceed with the next one. This means that no further action or rule will be executed for the current record and the first rule of the program will be executed for the next record. E.g.

```
#!/usr/bin/awk -f

# Print first name
{ print $1 }

# skip if not letters B and M
! (/^B/ || /^M/) {
       print "␣␣␣Skipping"
       next
}

# print number and empty line
{
  print $2
  print ""
}
```

6_patterns_actions_variables/ex_next.awk

[2] See example 8.8 in [1].

```
$ 6_patterns_actions_variables/ex_next.awk resources/data/telephonelist
```

```
Amelia
␣␣␣Skipping
Anthony
␣␣␣Skipping
Becky
555-7685

Bill
555-1675

Broderick
555-0542

Camilla
␣␣␣Skipping
Fabius
␣␣␣Skipping
Julie
␣␣␣Skipping
Martin
555-6480

Samuel
␣␣␣Skipping
Jean-Paul
␣␣␣Skipping
```

Exercise 6.5. In the optional part of exercise 5.5 on page 47 we realised that one instrument was a bit off. We will now write a script which aids with automatically excluding data from an apparatus with strange behaviour.

- Copy your script from ex. 5.5.
- Modify it such that the data of an apparatus is only considered if the average of its measurements is below `-0.05`.
- In other words: Neither should the printing of the computed average occur nor should the data be considered for the total average in the end. Instead you might want to print a nice message which says that the apparatus was excluded from the final average.
- The easiest way to achieve this is by first computing the average and to skip everything else if this is below the given threshold.

6.2.3 if-else statement

The `if`-`else` statement in `awk` has the syntax

```
if (condition) then_action else else_action
```

or alternatively

```
if (condition) then_action
```

since the `else`-part is optional. If `condition` is “true” the `then_action` is executed, otherwise the `else_action` is executed, so it exists.

- The `condition` may be pretty much any `awk` statement. It is only considered to be “false” if it evaluates to `0` or the empty string. Usually checks involving the conditional operators (see section 3.5.3 on page 25) are used for `condition`. E.g.

```
if (x % 2 == 0)
  print "x␣is␣even"
else
  print "x␣is␣odd"
```

- If more than one action or `awk` statement should be executed as part of `then_action` or `else_action` the statements have to be grouped together using curly braces (`{`. . . `}`) E.g.

```
if ($2 ~ /some.regex/) {
  print "Second␣field␣matches."
  print "Hooray!"
  # Increment x
  x++
} else {
  print "Oah␣no␣match"
  # decrement:
  --x
}
```

- There also exists a special kind of `if` statement when dealing with `awk` arrays. This is discussed separately in section 7.1 on page 64.

6.2.4 while statement

If one wants to execute certain actions repetitively for some number of times, one needs to employ a so-called **loop**. The `while` statement in `awk` is a particular simple looping statement:

```
while (condition)
  loop_action
```

It causes `loop_action` to be executed over and over again until `condition` is no longer “true”.

- The rules for the `condition` expression to be considered “true” or “false” are exactly the same as with the `if` statement above.

- The `condition` is both tested before the first `loop_action` is executed and after each execution of `loop_action`. If it is no longer true the loop is exited and program execution continues below the loop.
- If many actions are executed in the `loop_action`, we need to supply surrounding braces (`{...}`) again.

Example 6.6. This `awk` program computes for each record the n-th element of the famous Fibonacci sequence, where n is supplied as the first field of the record.

```awk
#!/usr/bin/awk -f
{
  # Extract n and convert to number
  n=+$1

  # The previous element of the sequence
  prev=0

  # The result we want to finally print
  res=1

  # Exit on an error with exit code 1:
  if (n <= 0) {
    print "Error:␣n␣<=␣0:␣" $1
    exit 1
  }

  # Special case: Just print the value and jump to next record
  if (n == 1) {
    print prev
    next
  }

  # Decrement n by 2 (the first element of the sequence was
  # dealt with above and the second requires no computation)
  n -= 2

  # Compute Fibonacci and decrement n as we go along
  while (n > 0) {
    tmp = res+prev
    prev = res
    res = tmp
    n -= 1
  }
  print res
}
```

6_patterns_actions_variables/ex_fibonacci.awk

```
$ echo -e "1\n3\n9\n30" | 6_patterns_actions_variables/ex_fibonacci.awk
```

```
0
1
21
514229
```

Exercise 6.7. Repeat the idea of the previous example:

- For each record compute (and print) the factorial $n!$, where n is the first field of the record. You may assume that this is a positive number.
- Try to compute a couple of fairly large factorials like 50! or 100!. How does `awk` deal with this?

6.2.5 `for` statement

The `for` statement makes it more convenient to use loops which perform some sort of counting. Its general form looks like

```
for (initialisation; condition; increment)
  loop_action
```

- Once this statement is encountered, first the `initialisation` statement is executed.
- Then `condition` is checked. If it is "true", `loop_action` and `increment` are executed, followed by another check for `condition`. This `loop_action`-`increment`-`condition` sequence continues until the `condition` is "false".
- In other words the `for` statement is just a shorthand for

```
initialisation
while (condition) {
  loop_action
  increment
}
```

- E.g. in order to print something 5 times we could use[3]

```
for(c=1; c <= 5; ++c)
  print "something"
```

[3] In `awk` we conventionally count 1-based, i.e. unlike many other programming languages counting goes like $1, 2, 3, \ldots$ and not $0, 1, 2, \ldots$.

- More complicated increments and bodies involving multiple lines are allowed as well (if surrounded by braces).

```
#!/usr/bin/awk -f
{
  for (i = 1; i <= 100; i *= 2) {
    print(i, $0)
  }
}
```

6_patterns_actions_variables/ex_for_complicated.awk

- All three statements in the brackets after `for` are optional. If nothing needs to be done in the relevant step, it may be empty, e.g.

```
  for(;;)
    print("bla")
```

  is an infinite loop or

```
  c = 15
  for(;c <= 30;++c)
    ++c
```

  just increments `c` until it is `31`.

- Note that initialising more than one variable in the `initialisation` statement or incrementing more than one variable in `increment` are not allowed[4].

- There also exists a special *for each* type of loop, which deals with elements of an `awk` array. See section 7.1 on page 64 for more details.

The variables `$i`, which hold the content of the fields, are different from other variables because the value after the `$` may in fact be computed by an arbitrary expression. E.g. the expression `$(5+3)` returns the content of the 8th field. One often exploits this together with `for` loops in order to do something for all fields of a records without knowing exactly how many fields there are.

Example 6.8. The `awk` program below computes the sum of all fields of a record and prints it:

```
#!/usr/bin/awk -f
{
  sum=0         # initialise the sum

  # Loop over all fields:
  # NF is a special variable (see next section) which contains the
  # number of fields of the current record
  for(i=1; i<=NF; ++i) {
    sum += $i   # add the value of the ith field
  }

  print sum     # print the result
}
```

6_patterns_actions_variables/ex_for_fields.awk

[4] Unlike `for`-loops in `C++`

It works as expected:

```
$ echo -e "4␣5␣6␣1\n␣1.1␣-1.1␣0␣9.1" | ↙
    ↪6_patterns_actions_variables/ex_for_fields.awk
```

```
16
9.1
```

Exercise 6.9. Redo exercise 6.7 on page 57 using a `for` loop.

Exercise 6.10. Take another look at your script from exercise 5.5 on page 47. Generalise it such that an arbitrary and potentially different number of measurements may be taken with each apparatus.

6.2.6 `break` statement

The `break` statement can be used to jump out of the innermost loop in encloses. For example the program

```
#!/usr/bin/awk -f
{
  # Accumulate the divisors
  divsum=0

  # Loop over all numbers excluding the
  # one we deal with:
  n=$1
  for (i=1; i < n; ++i) {
    if (n % i == 0) {
      divsum+=i
      if (divsum > n) break
    }
  }

  if (divsum == n) {
    printf("%d␣is␣a␣perfect␣number\n",n)
  } else {
    printf("%d␣is␣not␣a␣perfect␣number\n",n)
  }
}
```

6_patterns_actions_variables/ex_break.awk

finds out whether the first field of each record is a perfect number[5]. For example

```
$ echo -e "6\n19\n28\n33550336" | ↙
    ↪6_patterns_actions_variables/ex_break.awk
```

```
6␣is␣a␣perfect␣number
19␣is␣not␣a␣perfect␣number
```

[5] A perfect number is a number, for which the sum of all its proper divisors is equal to the number itself. E.g. 6 is a perfect number, since $1 + 2 + 3 = 6$. So is 28, because $1 + 2 + 4 + 7 + 14 = 28$.

```
28␣is␣a␣perfect␣number
33550336␣is␣a␣perfect␣number
```

Exercise 6.11. Write an `awk` program, which checks for each record whether the first field is a prime number. If this is not the case the program should print the smallest (non-trival) divisor of the number. Try it on a few cases (e.g. 101, 1001)

6.2.7 `continue` statement

The `continue` statement on the other hand may be used inside a loop to skip the remaining statements of the current loop cycle. E.g.

```
#!/usr/bin/awk -f

{
  for (i=0; i<$1; ++i) {
    print "Reached␣1␣for␣" i
    if (i%2==0) continue
    print "␣␣␣␣␣␣␣␣2␣for␣" i
  }
}
```

6_patterns_actions_variables/ex_for_continue.awk

```
$ echo "5" | 6_patterns_actions_variables/ex_for_continue.awk
```

```
Reached␣1␣for␣0
Reached␣1␣for␣1
␣␣␣␣␣␣␣␣2␣for␣1
Reached␣1␣for␣2
Reached␣1␣for␣3
␣␣␣␣␣␣␣␣2␣for␣3
Reached␣1␣for␣4
```

6.3 Builtin and special variables in `awk`

Apart from the variables you assign and alter in your program, `awk` provides a few other variables which have special meaning. Some of these change the way `awk` behaves and some just provide access to information which may be useful to your program.

This section just gives an overview of the most important of these special variables. The full list can be found in section 7.5 of the `gawk` manual [2].

These variables change `awk`'s behaviour when parsing data:

`RS` The record separator. Changes how input data is split into records. See section 4.1 on page 33 for details.

`FS` The field separator. Changes how records are split into fields. See section 4.2 on page 34.

`ORS` Changes the extra character `print` appends to the printed data. See section 5.1.1 on page 41.

`OFS` Changes how the arguments supplied to `print` are separated in the output and defines the character between the fields once `$0` is rebuilt. See section 5.1.1 on page 41.

`IGNORECASE` If this variable is set to any value other than `0` all string comparisons and regular expression matching operations are case independent. Thus the comparison operators `~` and `!~` as well as the regular expression versions of `RS`, `FPAT` and `FS` all ignore case when doing their particular operation.

The following variables contain information, mostly of parsed data.

`NR` The number of input records parsed so far. Its value is incremented whenever a new record is touched, i.e. it has value 1 when the rules are considered for the first record, value 2 when the second record is worked upon and so on. E.g. the program

```
#!/usr/bin/awk -f
{ printf("%4d␣␣%s\n",NR,$0) }
```

6_patterns_actions_variables/ex_line_nr.awk

adds the line number to the input data and

```
#!/usr/bin/awk -f
END { print NR }
```

6_patterns_actions_variables/ex_line_count.awk

just prints the number of lines in a file.

`NF` Contains the number of fields in the current record. See example 6.8 on page 58 for details.

`$0` Variable containing the full record. See section 3.2 on page 14.

`$1, $2, ...` Variable containing individual fields. The value after the `$` may be computed from an expression. See section 3.2 on page 14 as well as 6.2.5 on page 57 for details.

Exercise 6.12. In this exercise we want to collect some statistical data about the usage of the vowels "`a`" and "`e`" in a book of Project Gutenberg. Take the book `resources/gutenberg/pg5200.txt` as an example. Your program should print both the total number of characters, the number of times "`a`" as well as the number of times "`e`" have been encountered. Some hints:

- Recall that setting `FS=""` causes each character to be a field on its own.
- Use a `for`-loop to go over each field in each record.
- Keep in mind that the beginning and the end of a project gutenberg file are *not* actually part of the book. See exercise 6.3 on page 51.

Chapter 7

Arrays

So far we only dealt with simple variables, which were able to store a single value — either integer or floating point or string. awk also supports so-called **arrays**, i.e. tables of values, which can be accessed by a common name and an associated index. For example the code

```
array[0] = 15
array[1] = "dog"
array[10] = "number␣10"
```

creates an awk array called array and stores three values inside it, namely the integer 15 under the index 0, the string "dog" under index 1 and the string "number␣10" under 10. We can refer to these stored values by exactly the same syntax, e.g.

```
print "My" array[1] "␣" array[10] "."
```

would print "My dog number 10.". Some properties of awk arrays:

- The number of elements which is to be stored in the array does not need to be known, neither does it need to be constant during the program run.
- The index range does not need to be consecutive in any kind. But of course you *could* use consecutive integers like $1, 2, \cdots 100$ if you wanted.
- You may use strings as indices for awk arrays, e.g.

  ```
  #!/usr/bin/awk -f
  {
    arr["first"] = $1
    arr["second"] = $2
  }
  END {
    print("first:",arr["first"])
    print("second:",arr["second"])
  }
  ```

  7_arrays/ex_stringindices.awk

```
$ echo 3 4 | 7_arrays/ex_stringindices.awk
```

```
first:␣3
second:␣4
```

- In fact any index which is supplied to an array is automatically converted to a string. Even the integers in the introductory example[1].
- Using non-present indices will automatically generate an empty entry, which may then be interpreted as zero or the empty string.

```
#!/usr/bin/awk -f

{
  # Entry does not exist for the first record
  # => acts as zero
  arr["some"]+=$1
}
END {
  print "We␣have␣" arr["some"] "␣and␣-->" arr["any"] "<--"
}
```

7_arrays/ex_empty.awk

```
$ echo 3 | 7_arrays/ex_empty.awk
```

```
We␣have␣3␣and␣--><--
```

- Overall `awk` arrays resemble more features of what other programming languages call a **map** or a **dictionary**.
- It is important to keep in mind, that `awk` keeps a central register of all user-defined names. This means that there cannot be a normal variable and an array of the same name. Next to variables and arrays there is one further thing, namely user-defined functions (see section 8.2 on page 74) for which the same applies: If a name is taken by one of these structures, there cannot be another user-defined thing under the same name, even it is a different kind.

Example 7.1. Suppose we have a file like

```
2   line two
5   line seven
4   line six
1   line one
3   line four
```

7_arrays/unsorted.in

where the first column gives the proper sorting.

[1] This has subtle consequences, see 8.3 in [2] for details.

The program

```
#!/usr/bin/awk -f
{
  # store record in array
  # indexed under first column
  array[$1] = $0

  # determine and store maximum index
  if ($1 > max) {
    max = $1
  }
}
END {
  # Print array entries
  # Here we assume that all values from 1 until max
  # are actually used by at least one row in the input.
  for(x=1; x<=max; ++x) {
    print array[x]
  }
}
```

7_arrays/ex_sort.awk

represents a very naive sorting algorithm for such data. It yields

```
1␣␣␣line␣one
2␣␣␣line␣two
3␣␣␣line␣four
4␣␣␣line␣six
5␣␣␣line␣seven
```

as it should.

7.1 Statements and control structures for arrays

In the example above we had to assume that the first column of the input data contains each integer from 1 to some kind of maximum at least once. One can lift this assumption using the statement

```
if (index in array)
  body
```

which is a special version of the `if` statement. It checks whether an index has been used in an array, i.e. that `array[index]` is an existing element in the array. In order to generalise our code, we just have to insert such a check in the `END` rule:

```
#!/usr/bin/awk -f
{
  # store record in array
  # indexed under first column
  array[$1] = $0

  # determine and store maximum index
  if ($1 > max) {
    max = $1
  }
}
END {
  # Print array entries
  for(x=1; x<=max; ++x) {
    if (x in array) {
      print array[x]
    }
  }
}
```

7_arrays/ex_sortgen.awk

Another common task when dealing with arrays is to scan over all contained elements and do something with them. Since indices in `awk` can be any string it is a bad idea to systematically generate all possible indices (i.e. strings) and just retrieve the values for those which exist in a given array. Instead `awk` provides the statement

```
for (index in array)
  body
```

which loops over all *indices* of an array. So using the subscript statement `array[index]` we can then retrieve all values inside the loop. We can hence perform a task *for each* element of the array. This is why this type of loop is also called a for-each loop. Note, that there is no guarantee *in which order* the indices are traversed by such a loop[2].

Example 7.2. We want to write an `awk` program that prints the distribution of values in the third column of `resources/data/telephonelist`. One way to do this is

```
#!/usr/bin/awk -f

{
  # if this value does not exist
  # store 1, else increment:
  count[$3]++
}

END {
  # print table summarising what values we have
  # and how many times they occurred:
  print "value␣|␣count"
  print "------+------"
  for (val in count) {
    printf("%5d␣|␣%5d\n",val,count[val])
  }
}
```

7_arrays/ex_for_each.awk

which gives the output

[2] There are ways to change this, see section 8.1.6 in [2].

```
value␣|␣count
------+------
␣␣␣␣0␣|␣␣␣␣␣1
␣␣␣␣1␣|␣␣␣␣␣3
␣␣␣␣2␣|␣␣␣␣␣3
␣␣␣␣3␣|␣␣␣␣␣2
␣␣␣␣4␣|␣␣␣␣␣1
␣␣␣␣5␣|␣␣␣␣␣1
```

Finally it is possible to remove values from an array using the `delete` statement:

```
#!/usr/bin/awk -f
{
  arr[$1] = "value"
  arr["blu"] = "ber"

  if ("blu" in arr) {
    print "blu␣exists␣and␣has␣value␣" arr["blu"]
  } else {
    # Should not happen
    print "Cannot␣happen"
  }

  delete arr["blu"]

  if ("blu" in arr) {
    # Should not happen
    print "blu␣still␣exists??␣with␣value␣" arr["blu"]
  } else {
    print "No␣blu␣no␣more"
  }

  if ($1 in arr) {
    print "We␣still␣have␣" $1 ":␣arr[" $1 "]=" arr[$1]
  }
}
```

7_arrays/ex_delete.awk

```
$ echo "blubber" | 7_arrays/ex_delete.awk
```

```
blu␣exists␣and␣has␣value␣ber
No␣blu␣no␣more
We␣still␣have␣blubber:␣arr[blubber]=value
```

A whole array `array` at once can be deleted by

```
delete array
```

Exercise 7.3. We want to generalise exercise 6.12 on page 61, such that not only the all “a”s and “e”s, but all characters are counted. Your program should print both the total number of characters as well as the use count of all characters as a nicely formatted table.

Some Hints:

- Be very careful to only include characters of the book in the counts.
- Use an array to perform the counting of the characters.

(optional) Try to sort your output according to the count using an external shell command.

7.2 Multidimensional arrays

Some kind of multidimensional arrays exist in `awk` as well. We do not want to discuss the details here (see 8.5 in [2]), but just show a small example. The program

```
#!/usr/bin/awk -f
{
  # Determine maximum number of fields
  # (i.e. the maximum number of columns
  if (NF > max_nf) max_nf = NF

  # Store values of this record(==row) away
  for (i=1; i<=NF; ++i) {
    # store transposed values in a 1-based 2d array
    values[i,NR] = $i
  }
}

# print the resulting array:
END {
  for (i=1; i<=max_nf; ++i) {
    for(j=1; j<=NR; ++j) {
      printf("%s␣",values[i,j])
    }
    printf "\n"
  }
}
```

7_arrays/ex_multidim.awk

transposes a matrix of values. E.g. given the input

```
1 2 3 4 5
4 3 2 1 5
3 1 2 5 4
5 3 4 2 1
```

7_arrays/values.mat

it produces

```
1␣4␣3␣5
2␣3␣1␣3
3␣2␣2␣4
4␣1␣5␣2
5␣5␣4␣1
```

Chapter 8

Functions

In most programming languages a **function** is a small snippet of code, which is given a name. Once such a function has been **defined**, i.e. the association from the **function name** to the snippet of code has been made, we may also use the function name in order to refer to said code. One says we **call** the function.

In other words functions are a convenient feature in order to make code more reusable. We only need to write it once, namely in the function definition, and then we can call it from many places in the program.

Data flow between the outside program and the inner function is sometimes required. This is why functions may take one or more **arguments**, i.e. values or variable names which are to be specified when calling the function. These are then passed on to code which the function name refers to. This code can do its trick with the argument values and optionally **return** resulting data back to the calling party.

For example consider the function `length`, which is already pre-defined in `awk`. This function takes a string as an argument and returns the number of characters of this string, i.e. its length. For example we can use it to determine the length of each field:

```
#!/usr/bin/awk -f
{
  for (i=1; i<=NF;++i) {
    $i = length($i)
  }
  print $0
}
```

8_functions/ex_length.awk

```
$ echo "blubber␣blubi␣bla␣di␣bla" | 8_functions/ex_length.awk
```

```
7␣5␣3␣2␣3
```

Notice, that functions are called by providing their arguments in a comma-separated list inside parenthesis, i.e.

```
function(arg1, arg2, ...)
```

8.1 Important built-in functions

Before we discuss how to define your own functions in section 8.2, we will first take a look at those functions `awk` already provides for our convenience. A lot of common tasks like string substitution, splitting strings or just computing the sine or retrieving a random number are already implemented by `awk` in the form of built-in functions.

This section only gives an overview of the most important functions for numeric computations (section 8.1.1) and for manipulating strings (section 8.1.2 on the next page). Many of these functions seem fairly complicated and you will probably need a little practice to make the most out of them. For the first reading it is probably best to just take mental note what kind of functionality is available. Later, when you write a program and require one of these functions, you can still refer back to this section or to section 9.1 of the `gawk` manual [2] for detailed information.

8.1.1 Numeric functions

The following numeric functions are predefined in `awk`. Optional arguments are denoted in square brackets (`[ ]`).

`atan2(y, x)` Return the arcus tangent of $\frac{y}{x}$ in radians. For numerical reasons it is desirable to take `y` and `x` as separate arguments[1].

`cos(x)` Return the cosine of `x` in radians.

`exp(x)` Return the exponential of `x`.

`int(x)` Return the nearest integer to `x`, truncated towards zero. In other words `int(2.8)` is 2 and `int(-2.8)` is `-2`.

`log(x)` Return the natural logarithm of `x`.

`rand(x)` Return a random number, uniformly distributed in `[0,1)`, i.e. the value could be zero, but is never one. Do not rely on good randomness here, some `awk` implementations have pretty bad random number generators. `gawk` seems to be ok here. Furthermore most `awk` implementations (including `gawk`) start with the same seed each execution, i.e. will produce the same sequence of random numbers each time. In other words a program like

```
#!/usr/bin/awk -f

{
  for(i=0; i<$1; ++i) {
    printf("%12.10f␣", rand())
  }
}
```

8_functions/ex_random.awk

will print the same numbers each time.

[1] See `https://en.wikipedia.org/wiki/Atan2` for details.

```
$ echo 3 | 8_functions/ex_random.awk
```

```
0.2377875122␣0.2910657359␣0.8458138536
```

`sqrt(x)` Compute the square root of `x`.

`srand([x])` Provide a seed for `rand()`. If no arguments are provided, i.e. the function is called like `seed()`, the current date and time is used as seed. For example the following script prints the result of simulated dice rolls:

```
#!/usr/bin/awk -f

BEGIN {
  srand() # Seed with current date and time
}

# function to roll a dice (see section 8.2 for more
# explanation how functions are defined)
function roll_dice() {
       return 1 + int(rand()*6)
}

{
  for(i=0; i<$1; ++i) {
    printf("%d␣", roll_dice())
  }
}
```

8_functions/ex_rolldice.awk

```
$ echo 3 | 8_functions/ex_rolldice.awk
```

```
3␣6␣4
```

`sin(x)` Return the sine of `x` in radians.

Note that `awk` does not have a function like `abs` which would return the absolute value of a number. We will define such a function in section 8.2 on page 74.

8.1.2 String functions

The following gives a list of important string functions. Optional arguments are again denoted in square brackets (`[ ]`).

`asort(source [ ,dest ])`
Sort the values of `source` and replace the indices by with sequential integers starting with one.[2] Return the number of elements in `source`. If `dest` is provided, the sorted array is written into the variable `dest` and `source` remains as it was. For example

[2] This function actually has an optional third argument, controlling how sorting is done, see section 12.2.2 in [2] for details.

```
#!/usr/bin/awk -f
BEGIN {
  a["array"] = "awk"
  a["value"] = "is"
  a["some"] = "cool"

  res = asort(a,b)
  print(res, b[1], b[2], b[3])
  print("a", a["some"], a["array"], a["value"])
  print("")

  res2 = asort(a)
  print(res2, a[1], a[2], a[3])
}
```

8_functions/ex_asort.awk

```
3␣awk␣cool␣is
a␣cool␣awk␣is

3␣awk␣cool␣is
```

`asorti(source [ ,dest ])`
Works exactly like `asort`, but sorts indices instead of values, i.e.

```
#!/usr/bin/awk -f
BEGIN {
  a["array"] = "awk"
  a["value"] = "is"
  a["some"] = "cool"

  res = asorti(a,b)
  print(res, b[1], b[2], b[3])
  print("a", a["some"], a["array"], a["value"])
  print("")

  res2 = asorti(a)
  print(res2, a[1], a[2], a[3])
}
```

8_functions/ex_asorti.awk

```
3␣array␣some␣value
a␣cool␣awk␣is

3␣array␣some␣value
```

`length([string])` Return the number of characters of a string or the number of elements in an array. If `string` is not provided, the number of characters in `$0` is returned instead.

`gensub(regex, replacement, how [ ,target ])`

Search through the `target` for matches of the regular expression `regex` and replace one or more occurrences by `replacement`. The variable `how` controls how many replacements are done. The original string `target` is *not* altered as the function *returns* the modified string.

If `target` is absent, the function operates on the current record, i.e. `$0`. The `regex` may either be a regex string like `":"` or `"[a-z]"` or a regular expression literal like `/:/` or `/[a-z]/`. E.g. the following call replaces the first "`a`" in the current record

```
$ echo "a␣b␣c␣a␣b␣c" | awk '{ print gensub(/a/, "AA",1) }'
```

```
AA␣b␣c␣a␣b␣c
```

as does this

```
$ echo "a␣b␣c␣a␣b␣c" | awk '{ print gensub("a", "AA",1, ↙
    ↪$0) }'
```

```
AA␣b␣c␣a␣b␣c
```

"`how`" is either a string or a number. If it is a string starting with the letter `"g"` or `"G"` (for global) all occurrences of the matching `regex` are replaced. If it is a number then exactly this occurrence is replaced. E.g.

```
$ echo "a␣b␣c␣a␣b␣c" | awk '{ print gensub(/a/, "AA",2) }'
```

```
a␣b␣c␣AA␣b␣c
```

or

```
$ echo "a␣b␣c␣a␣b␣c" | awk '{ print gensub(/a/, ↙
    ↪"AA","g") }'
```

```
AA␣b␣c␣AA␣b␣c
```

Some characters in the replacement string are special: They can be used to refer back to the precise parts of the target string, which are matched by a grouping expression in the regex[3]. See 9.1.3 in [2] for more details. Just one example: This program swaps the first two words of the record using the groupings like "`([^ ]+)`" and the back references "`\\1`" and "`\\2`":

```
$ echo "word1␣word2␣word3␣word4" | awk '{ print ↙
    ↪gensub(/([^ ]+) ([^ ]+)/, "\\2␣\\1",1,$0) }'
```

```
word2␣word1␣word3␣word4
```

[3] This is exactly as in the `sed` command `s`.

`split(string, array [ ,fieldsep ])`

Split `string` into pieces at every character which matches the regular expression `fieldsep`. Store the pieces into `array` indexed from 1 to n, where n is the value returned by `split`, i.e. `array[1]` contains the first piece, `array[2]` the second piece and so on.

`fieldsep` may be either a regex string like `":"` or `"[a-z]"` or a regular expression literal like `/:/` or `/[a-z]/`. If this parameter is missing the value of `FS` is used. If it is present `FS` has no influence on `split`.

For example:

```
#!/usr/bin/awk -f
BEGIN {
  string="ooh-ya-koo"

  n=split(string,a,"-")

  print("string␣is␣split␣into", n, "elements:")
  for(i=1;i<=n;++i) print("␣␣:" a[i] ":")

  m=split(string,a,/[hky]/)
  print("string␣is␣split␣into", m, "elements:")
  for(i=1;i<=m;++i) print("␣␣:" a[i] ":")
}
```

8_functions/ex_split.awk

```
string␣is␣split␣into␣3␣elements:
␣␣:ooh:
␣␣:ya:
␣␣:koo:
string␣is␣split␣into␣4␣elements:
␣␣:oo:
␣␣:-:
␣␣:a-:
␣␣:oo:
```

Note, that `split` has another optional argument for storing the actual separator characters it encountered. See 9.1.3 in [2] for further details on this.

`substr(string, start [ ,length ])`

Returns a substring of `string` which is `length` characters long and starts at character number `start`. In `awk` the first character has index number 1[4]. If `length` is missing all characters of `string` starting at `start` are returned. Examples:

```
$ echo "bladi_bla␣4" | awk '{ print substr($1,$2) }'
```

```
di_bla
```

or

[4] This is different from many other programming languages like `C`, `C++` or `python`, where the first character has index `0`!

```
$ echo "bladi_bla␣2" | awk '{ print substr($1,$2,4) }'
```

```
ladi
```

`tolower(string)` Return a copy of `string` with each uppercase character replaced by its lowercase equivalent. Non-alphabetic characters are not touched.

```
$ echo "somE4CRaZY_strINg" | awk '{ print tolower($0) }'
```

```
some4crazy_string
```

`toupper(string)` Return a copy of `string` with each lowercase character replaced by its uppercase equivalent. Non-alphabetic characters are not touched.

```
$ echo "somE4CRaZY_strINg" | awk '{ print toupper($0) }'
```

```
SOME4CRAZY_STRING
```

8.2 User-defined functions

A user-defined function looks like this

```
function name(parameter_list) {
  body
}
```

Such a definition may not be placed inside an action block or a pattern, but apart from that anywhere in the code. Since `awk` reads the full code before executing it, the function may even be defined *after* its first use. Conventionally one places all functions before specifying the first rule of the program.

- The `name` of a function follows the same rules as the name of a variable: It is a sequence of letters, digits and underscores, that does not start with a digit.
- The `parameter_list` is a comma-separated list of variable names. Inside the `body` we may use these variables to refer to the values of the arguments, which were passed to the function when it was called.
- `body` may contain any `awk` statement, which may also be placed in an action block, e.g. other function calls, prints, arithmetic or comparison with variables.
- Inside a function body the special statement `return` exists

  ```
  return expression
  ```

  This may be used to return the value of `expression` back to the caller. The `expression` argument is optional.

  In either case: As soon as the `return` statement is encountered, the rest of the function body is ignored and the execution of the `awk` program continues at the place where the function was called.

- This description only provides a brief and non-exhaustive introduction into user-defined functions in `awk`. More information can be found in section 9.2 of [2].

Example 8.1. We want to define and use a `abs` function in order to compute the absolute value of an argument. The following program for example computes the absolute value of each field:

```
#!/usr/bin/awk -f
function abs(x) {
  if (x < 0) {
    return -x
  }
  return +x
}
{
  for (i=1; i<NF; ++i) {
    $i = abs($i)
  }
  print $0
}
```

8_functions/ex_abs.awk

```
$ echo "3␣-4␣-3.4␣19" | 8_functions/ex_abs.awk
```

```
3␣4␣3.4␣19
```

One could use a user-defined function like `abs` in a pattern. This program, for example, does only print those transfers of `resources/data/money.csv` which cause a balance change larger than 15.

```
#!/usr/bin/awk -f
function abs(x) {
  if (x < 0) {
    return -x
  }
  return +x
}

# Change Field separator (we need a comma)
BEGIN { FS="," }

# if no comment and balance change > 15
/^[^#]/ && abs($2) > 15 {
  printf("%-20s␣%.2f\n",$1,$2)
}
```

8_functions/ex_blance_change.awk

```
Transfer␣to␣8879␣␣␣␣␣-87.30
Transfer␣from␣2299␣␣␣-19.88
```

Example 8.2. The program

```
#!/usr/bin/awk -f

# A helper function to achieve the reversal
function rev_helper(string,start) {
  # start starts off to be the string length
  # (last character) and is reduced during the
  # recursive call. If it is 0 recursion has to end
  if (start == 0) return ""

  # Append the character under start and call
  # myself recursively reducing the value of start by one.
  return (substr(string,start,1) rev_helper(string,start-1))
}

# Do the reversal: Call the helper function
# appropriately
function rev(string) {
  return rev_helper(string,length(string))
}

{ print rev($0) }
```

8_functions/ex_rev.awk

reverses each record characterwise

```
$ echo "Some␣words␣and␣some␣blafasel." | 8_functions/ex_rev.awk
```

```
.lesafalb␣emos␣dna␣sdrow␣emoS
```

Exercise 8.3. *(demo)* Write an `awk` program which reads a matrix of input data like

```
16 26 88 6 10
1996 1101 -12 -582
0 225 -198776 -582
-5924 100 512 -582 10
16 16 2 -10
```

8_functions/matrix.data

and determins both the maximum element of the matrix as well as the maximum of the absolute values of all the matrix elements.
Note: There are many ways to solve this exercise. One particularly nice solution exploits the features of `awk` properly and manages to do this without any control structure like `if`, `for` or `while` whatsoever.

Chapter 9

Writing practical awk programs

This chapter both summarises some hints and tips for writing useful `awk` programs and gives a lot of examples. Most of these are not of great practical use by themselves, but show rather nicely what *could* be done with `awk`. Think of them as a source for inspirations for your own future `awk` programs.

9.1 When and how to use awk

In my experience `awk` is not a good general-purpose scripting language. It is, however, very good at processing line-based data files or more generally files, which can be somehow split up into records. Note, that most plain text files, which have some sort of a structured format, fall in this category. If one needs to deal with these files, often a few lines of `awk` code is enough do the trick. This is in contrast to other scripting languages like `bash` or `python`. These are less practical in such cases, merely because one has to write the code for parsing the data[1].

Therefore I usually use `awk` as a supplement to a main script, which overall orchestrates the work. Luckily `bash` and `awk`, for example, work together easily very well. For example `awk` variables can be set directly from the commandline, i.e. effectively from an outer main script. The respective flag is `-v`, which needs to be passed to the `awk` executable as such:

```
$ awk -v "name=value" ' awk_source '
```

For example

```
$ awk -v "var=VALUE" 'BEGIN { print var }'
```

```
VALUE
```

This way `awk` programs can usually be placed inline, i.e. entirely inside the `bash` script. For more details about combining `awk` and shell scripts see chapter 8 of [1].

[1] Please correct me if I am wrong here.

9.2 Re-coding Unix tools using awk

The basic design philosophy behind UNIX[2] implies that most UNIX utilities are small building blocks for achieving larger tasks. Therefore being able to re-code these utils with awk is a really helpful skill, just because one will need similar coding techniques when writing more complicated awk programs.

Example 9.1. We want to re-code cut, which can be used to extract fields separated by one specific character, by default a tab. cut has a couple of commandline parameters, which change its behaviour. We have not yet discussed how to do argument parsing in awk scripts[3], so we will do the settings hard-coded in the BEGIN rule:

```
#!/usr/bin/awk -f
BEGIN {
  ### Settings
  # The field separator (cut flag -f)
  f="\t"
  # Which field should we extract (flag -d)
  d=3
  ### End settings

  # Copy field separator from settings:
  FS=f
}

# Only print the appropriate field.
{ print $d }
```

9_practical_programs/cut.awk

Example 9.2. Next we want to do tee: A program, which writes everything it gets on its standard input to output as well as to disk.

```
#!/usr/bin/awk -f
BEGIN {
  ### Settings
  file="teeout.log"
}

# Print to stdout and to file
{
  print > file
  print
}
```

9_practical_programs/tee.awk

[2] Each program does only one thing, but does this thing right.

[3] This is discussed in section 7.5.3 of [2].

Example 9.3. Now we want to code `tac`. This program prints all lines of input, but in reverse order.

```
#!/usr/bin/awk -f

# Write each line of text into an array
{ lines[NR] = $0 }

# In the end print them in reverse order
END {
  for (i=NR; i>0; --i) {
    print lines[i]
  }
}
```

9_practical_programs/tac.awk

Exercise 9.4. Here are some UNIX utilities you could try to code yourself using `awk`. See the respective `man` pages for more detailed explanation what the program does:

- `wc -w`: Print the number of words in the input text.
- `uniq -c`: Print only unique lines of text, but print the number of times the line occurred after another.
- `sort`: A program which sorts its input lines lexicographically.
- `egrep`: A program which prints only lines of text, which match a given regular expression. (For this you will probably need an enclosing shell script or you need to hard-code the pattern to search for)

9.3 Example programs

This section gives a couple of concluding example programs, which employ techniques we discussed in this course. Some of these programs are build upon ideas of chapter 11 of the `gawk` manual [2], where even more examples can be found as well.

Example 9.5. The program

```
#!/usr/bin/awk -f

BEGIN {
  ### Settings
  shift=13     # The shift to use (here 13 chars)


  # All letters of the alphabet in lower and upper case
  alphaLower="abcdefghijklmnopqrstuvwxyz"
  alphaUpper=toupper(alphaLower)

  # The number of letters in the alphabet:
  NL=length(alphaLower)

  # Use the shift variable and above strings to build an array
  # for translating each character of the alphabet:
```

```
  for (i=1; i<=NL;++i) {
    # first lower case
    from=substr(alphaLower,i,1)
    to=substr(alphaLower,(i+shift)%NL,1)
    trans[from]=to

    # now upper case
    from=substr(alphaUpper,i,1)
    to=substr(alphaUpper,(i+shift)%NL,1)
    trans[from]=to
  }

  # Make sure that we traverse the data character by character:
  FS=""

  # Make sure there are no characters added to the output:
  OFS=""
}

# Function to translate the characters:
# alters all characters known to the alphabet
function translate(c) {
  if (c in trans) {
    return trans[c]
  } else {
    return c
  }
}

# Translate character by character ie field by field
{
  for(i=1; i<=NF; ++i) {
    printf("%1s",translate($i))
  }
  # Finish the line:
  print ""
}
```

9_practical_programs/caesar.awk

can perform a Caesar cipher “encryption”[4]. By default it performs the infamous `ROT13`[5].

[4] See `https://en.wikipedia.org/wiki/Caesar_cipher` for more information

[5] See for example `https://en.wikipedia.org/wiki/ROT13`

Example 9.6. The program

```
#!/usr/bin/awk -f

{
  # Normalise the input record:
  # a) Remove punctuation (i.e. everything which is not
  #    a space char or an alphanumeric character)
  $0 = gensub(/[^[:blank:][:alnum:]-]/,"","g",$0)
  #
  # b) Make everything lower case:
  $0 = tolower($0)

  # Add each field (word) to the wordcount array, which keeps
  # track of how many times a word has been used and increment
  # the count by one.
  for (i=1;i<=NF;++i){
    wordcount[$i]++
  }
}

END {
  # Print the count followed by the words
  for (a in wordcount) {
    printf("%5d %s\n",wordcount[a],a)
  }
}
```

9_practical_programs/word_usage.awk

prints a list of all words, which were used in the input data. A count, how often they are used, is determined and printed as well.

Example 9.7. UNIX systems usually come shipped with a dictionary of English words located under `/usr/share/dict/words`. Each line in the dictionary only contains a single word. We want to use this dictionary to build a list of anagrams in the English language. A word is an anagram to another if both words contain the same letters, but in a different order, e.g. “cat” and “act”. Here is the program:

```
#!/usr/bin/awk -f

# This function normalises a dictionary word by sorting the
# constituent letters alphabetically. Repetitions are kept, ie
# if a letter occurs multiple times in the word, the normalised
# version will also have it this number of times.
# The effect of the function is therefore:
# litter -> eilrtt
# cat -> act
# act -> act
# i.e. all anagrams will result to the same normalised string
function normalise_word(word) {
  # Use split to make an array, which contains the characters
  # of the word as array elements:
  split(word,arr,"")

  # Now sort the array
  n = asort(arr)

  # Concatenate the characters again and return
  res=""
  for (i=1; i<=n; ++i) {
    res = res arr[i]
  }
  return res
}

# Sort the array according to its values and store inside sorted
# the order of the indices required to obtain the sorted array.
# i.e. sorting
#     a["a"] = "b"
#     a["n"] = "a"
# using this function would yield
#     sorted[1] = "n"
#     sorted[2] = "a"
function argsort(array, sorted) {
  i=0
  for (key in array) {
    sorted[++i] = array[key] "@@@" key
  }
  n = asort(sorted)
  for (i=1;i<=n;++i) {
    sorted[i] = gensub(/^.*@@@/,"",1,sorted[i])
  }
  return n
}

# Skip all possessives (words with 's at the end)
/'s$/ { next }

{
  # In the data array we want to store the list of anagrams.
  # So the current word (record) is appended
  # to the value which is indexed by the normalised key
  key = normalise_word($1)
  val = data[key]

  # append, but only if there is something:
  if (val != "") {
    data[key] = val "␣" $1
  } else {
    data[key] = $1
  }

  # Keep a count of how many anagrams we found for this
  # normalised key
  count[key]++
}

```

```
END {
  # Drop keys with a count below 2 (i.e. no anagrams)
  for (key in count) {
    if (count[key] < 2) {
      delete count[key]
    }
  }

  # Perform argsort on count, i.e. sort the keys of count in a
  # way that the values they refer to come out sorted
  # if we write data[sorted[i]] with i running sequentially.
  n = argsort(count,sorted)

  for (i=1; i<=n; ++i) {
    print (data[sorted[i]])
  }
}
```

9_practical_programs/anagrams.awk

Running it on /usr/share/dict/words yields an output similar to

```
...
palest␣pastel␣petals␣plates␣pleats␣staple
least␣slate␣stale␣steal␣tales␣teals
opts␣post␣pots␣spot␣stop␣tops
carets␣caster␣caters␣crates␣reacts␣recast␣traces
pares␣parse␣pears␣rapes␣reaps␣spare␣spear
```

Example 9.8. The final program sparsifies an mtx file[6] by dropping those matrix components which have an absolute value smaller than 10^{-12}. We no longer need to store those values, which implies that the number of entries in the mtx file decreases.

```
#!/usr/bin/awk -f

BEGIN {
  # Our threshold for the values
  eps = 1e-12
}

function abs(a) {
  if (a<0) {
    return -a
  } else {
    return +a
  }
}

# Print all comment lines
/^%/ { print; next }

# the first non-comment line is the shape of the matrix
# interpret and store it:
```

[6] See appendix B.1 on page 86 for more details.

```
shape_encountered == 0 {
  shape_encountered=1

  # number of rows and columns of the matrix:
  nrows = $1
  ncols = $2

  # Note: The d, i.e. the number of entries
  # stored in the mtx file, will change due
  # to this program and is hence not kept
  next
}

# All other comment lines contain data.
# Add it to the values array if its value
# (3rd field) is larger than the threshold
abs($3) > eps {
  # Store the record and increase non-zero count
  values[++nnonzeros] = $0
}

# In the end write the altered file:
END {
  # Print shape and number of non-zeros:
  print(nrows,ncols,nnonzeros)

  # Print all values:
  for (i=1; i<=nnonzeros; ++i)
    print(values[i])
}
```

9_practical_programs/sparsify_mtx.awk

Appendix A

Obtaining the files

In order to obtain the example scripts and the resource files you will need for the exercises, you should run the following commands:

```
1 # clone the git repository:
2 git clone https://github.com/mfherbst/awk-course
3
4 # download the books from Project Gutenberg
5 cd awk-course/resources/gutenberg/
6 ./download.sh
```

All paths in this script are given relative to the directory `awk-course`, which you created using the first command in line 2 above.

All exercises and example scripts should run without any problem on all Linux systems that have the GNU `awk` implementation (`gawk`) installed. If other `awk` implementations (see 1.2 on page 2) are used, it may happen that examples either do not work or give other output due to the differences between the different `awk` dialects.

Appendix B

Supplementary information

B.1 The `mtx` file format

The main idea of the `mtx` file format is to be able to store matrix data in a plain text file without storing those matrix entries which are zero. This is achieved by only storing a selection of the matrix components and defaulting all other component values to 0.

The `mtx` files we use in this course[1] for demonstration purposes, follow a very simple structure

- All lines starting with "`%`" are comments
- The first line is a comment line.
- The first non-comment line contains three values separated by one or more `<space>` or `<tab>` characters:
  - The number of rows
  - The number of columns
  - The number of entries, which are explicitly set in the file. We will refer to this number as d.
- All following lines — the explicitly provided entries — have the structure
  - Row index (starting at 1, i.e. 1-based)
  - Column index (1-based)
  - Value

  where the individual columns are again separated by one or more `<space>` or `<tab>` chars. The number of lines in this latter block and the number d provided on the first non-comment line have to agree in a valid `mtx` file.

All matrix components, which are not listed in the latter block, default to a value 0.

[1] We will only use a subset of the full format, which can be found under `http://math.nist.gov/MatrixMarket/formats.html#mtx`

Some examples

- Consider the file

```
1  %%MatrixMarket matrix coordinate real symmetric
2  3 3 9
3  1 1 1
4  1 2 1
5  1 3 1
6  2 1 2
7  2 2 2
8  2 3 2
9  3 1 3
10 3 2 3
11 3 3 3
```

resources/matrices/3.mtx

The first line is a comment line, which we can ignore. The second line tells us that the matrix represented is a 3×3 matrix and that all nine entries are provided in the Matrix Market file. Lines 3 to 11 then list the values. Overall this file represents the matrix

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}.$$

- The file

```
%%MatrixMarket matrix coordinate real
3 3 9
1 1 1
1 2 0
1 3 0
2 1 0
2 2 2
2 3 0
3 1 0
3 2 0
3 3 3
```

describes a 3×3 matrix as well, namely the diagonal matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}.$$

If we want to avoid storing the zeros, we can use the equally valid `mtx` file

```
%%MatrixMarket matrix coordinate real
3 3 3
1 1 1
2 2 2
3 3 3
```

Notice, how the last value in the first non-comment line has changed as well.

Bibliography

[1] Michael F. Herbst. Advanced bash scripting, August 2015. URL `http://docs.mfhs.eu/teaching/bash_course-ss-2015/notes.pdf`.

[2] Arnold D. Robbins. GAWK: Effective AWK Programming, April 2014. URL `https://www.gnu.org/software/gawk/manual/`.

Index

`;` (semicolon), 18
`awk` scripts, 29

action, 2, 13
action block, 49
alternation operator, 9
arguments, 68
arithmetic operators, 23
arrays, 62

Boolean comparison operators, 25
bracket expansion, 8

calling a function, 68
comma-separated value files, 34
comment character in `awk`, 16
complemented bracket expansion, 8

data-driven, 1, 4, 5
defining a function, 68

field separator, 13, 34
fields, 13
format control letter, 43
format modifier, 43, 44
format specifier, 43
function, 68
function name, 68

greedy, 38

Linux, 1
loop, 55

match, 6

pattern, 2, 6, 13, 49
POSIX character classes, 10
program flow, 49

record, 2, 6, 13, 49
record separator, 13, 33
regular expression, 1, 6
regular expression comparison operators, 26
regular expression operators, 7
return, 68
rule, 2, 13, 49

shebang, 29
strings in `awk`, 16

UNIX, 1

Variables in `awk`, 18

ways to run `awk`, 29