CS214 Project/*Projek* 2024
Braille translation/*vertaling*

Prof L van Zijl, I le Roux, M Keys

Feb 2024

Changelog

Date	Changes/amendments/clarifications
2024-03-18	Invalid content of input files: for brf files, you must check whether it only contains digits and dashes. For text files, you must check that it is not brf content (it must contain some alphabetical symbols).
2024-03-18	Mathematical signs: you only need to implement +, -, x, =. The brf for + is 235.
2024-03-18	Diacritics: those only need to be applied to vowels (a, e, i, o , u)
2024-03-18	Mathematical expressions: you only need to handle +, -, x and =, and do it symbol by symbol directly.
2024-03-18	Resolution of ambiguities added (see below – only expected from SH6).
2024-03-18	End of numbers: You may use the noncontract sign to resolve between 3a and 31, for the moment.
2024-03-07	Removed "Invalid argument type" error message, no longer required.

0.1 Resolution of ambiguities

When ambiguities are encountered in translating from Braille to text, you are supposed to translate the sentence with all valid possibilities. You first have to eliminate invalid translations, based on the word list. Then you have to list all the possible valid translations of the whole sentence, line by line.
For example, on Level 1.0 (uncontracted), consider the sentence
`Dit is afdeling 3a en nie afdeling 31 nie.`
The brf input (read as one line)
`6-145-24-2345 24-234 1-124-145-15-123-24-1345-1245 3456-14-1-2 1345-24-15`
`1-124-145-15-123-24-1345-1245 3456-14-1 1345-24-15-256` must then be translated back to text as four different lines in the output file:

```
Dit is afdeling 3a nie afdeling 3a nie.
Dit is afdeling 3a nie afdeling 31 nie.
Dit is afdeling 31 nie afdeling 3a nie.
Dit is afdeling 31 nie afdeling 31 nie.
```

Figure 1: An example of a Braille character for the symbol Z

Project brief

This project investigates automatic translation from Braille brf codes to text, and vice versa. This is of course only a lookup table if one Braille character corresponds to a single text symbol. There are some control characters (such as a character to indicate that the next symbol is uppercase). In addition, to make Braille shorter to read, some common words and morphemes are contracted into a single character, based on strict language rules. These contractions are classified into levels. Level 1 is uncontracted, and each subsequent level up to level 4, has sublevels. Each level provides a list of words/morphemes, and their contractions. So, you would have to develop a system that can translate both directions, for each sublevel. The project is divided into four phases; more details on each phase are given below.
In this project, you are required to run the system from the command line, with arguments to indicate an input file, a level, and the direction of translation. You must develop all code in Java, and may not use XML or other related language utilities (such as regular expressions). You may, however, use any of the other library code in the Sedgewick and Wayne stdlib and algs4 libraries.

Due dates/*Inhandigingsdatums*

Handin *Inhandiging*	Date *Datum*	Time *Tyd*	Expected *Verwagting*
Soft handin 1	23 Feb 2024	17h00	Directory structure
Soft handin 2	1 Mar 2024	17h00	Git pipeline, script marking, style
Soft handin 3	8 Mar 2024	17h00	Valid input, output
Soft handin 4	15 Mar 2024	17h00	Text to Braille and back
Soft handin 5	22 Mar 2024	17h00	Text, numbers
Phase 2 (midterm)	28 Mar 2024	17h00	Text, control chars, numbers, punctuation
Soft handin 6	19 April 2024	17h00	Contractions level 1
Soft handin 7	26 April 2024	17h00	Contractions level 2
Soft handin 8	3 May 2024	17h00	Contractions level 3
Final handin	6 May 2024	**13h30**	All phases
Demo, remarks	13 and 17 May 2024	14h00	

Each handin requires that the translation is possible in either direction. For example, for soft handin 4, you must be able to translate from text to Braille, and from Braille brf codes to text. The back translations can become quite tricky, as ambiguities may creep in. A brf file can be written as numerical values, separated by dashes and spaces, or it can be written as the text corresponding to those numerical values (excluding the dashes then).

Marking scheme

- 75% – Test cases (midterm handin, final handin).
- 10% – Code style and Javadoc comments (final handin).
- 15% – General impression / bonus marks for effort exceeding the standard specification (final handin).

The test cases are run for each phase, and the final mark for the test cases is calculated by the formula $0.20 * T_1 + 0.20 * T_2 + 0.30 * T_3 + 0.30 * T_4$, where T_i represents the results for the test cases of phase i

(see below for the different phases). The code style, comments, general impressions, and bonus marks are assigned once, after the handin of the last final phase. Note that a mark of 100% is the maximum you can obtain, regardless of bonus marks awarded.
It is important that you use the correct style throughout the development of the project, instead of a mad rush effort before the final handin date.
The general impression mark is given for modular design, object oriented design, consistency, use of constants, readability, consistency, correct usage of comments, and generisability. For example, do not use hard-coded values in your program.

Assessment

All marking will be done using scripts. It is therefore necessary that your project can be compiled and run from the command line. If it does not compile and run from the command line, you will receive zero for your project. This also means that you must follow the prescribed directory structure on git exactly.
When marking the project, various aspects and components will be tested separately. As such, the project must be able to run in different modes set via runtime arguments.
The input and output specification must be followed EXACTLY to facilitate script marking. Make sure that nothing except for the expected output is printed to standard output. Error message should be printed to the standard error stream.
A test script and some small examples will be provided through git for every stage of the project, so that you can test the correctness of the code as you progress. We will run exactly the same script to mark your project, but with a more extensive test bank of input examples.

Plagiarism

There is a plagiarism quiz on SUNLearn that you must complete. If you do not complete it, your project will not be marked. If your project is not marked due to a missing plagiarism quiz handin, you may request a late handin so that your project will be marked. However, you will then get a -5% penalty added to your final project mark. This rule holds for all handins (soft, midterm and final). Since there are ten handins in total, you could lose up to 50% in total due to a non-completion of the plagiarism quiz.
The plagiarism rules, and the consequences of breaking the rules, are explained below.

- No project code or text may be shared amongst individuals
- You may not show your code or solution to anyone. You many not share your code or solution on social media or via any electronic means. If you do so, you will receive zero if someone else submits it as their own.
- All submissions of the project code must be accompanied by the plagiarism quiz on SUNLearn. If you do not submit the plagiarism quiz, your submission will not be marked.
- Submissions will be checked for plagiarism against existing solutions on the internet, against Chat-GPT, against submissions of previous years, and against the other submissions in the class.
- If plagiarism is detected, both parties receive zero.

What to do, and what not to do, in terms of plagiarism avoidance

- We encourage discussions on solving a problem. You should discuss the concepts in general, but not on a detailed level. 'We worked together' is not an excuse for excessive collusion, as the assessments are all individual tasks. Include an acknowledgement if you exchanged ideas with a fellow student.
- You may not show your code or solution to anyone. You many not share your code or solution on social media or via any electronic means. If you do so, you will receive zero if someone else submits it as their own.
- You may use the libraries that accompany the text book (stdlib and algs4, excluding the language components such as regular expressions).

- You may use other external libraries only if those do not solve the core of the solution. You have to get the lecturer's permission to do so. You must include an acknowledgement of the use of such libraries at the top of your java file, as well as in the README.txt file for the project.
- Avoid the temptation to 'buy' code from a tutor service.
- You have to work regularly on the project, and submit your progress weekly. As we check weekly, that will help us to pick up early on too much collusion, and guide you to change your coding practices.
- In the case of suspected plagiarism, your soft handin progress on git will serve as proof that you did not (or did) commit plagiarism. Sudden replacement of a whole file with a totally different one, is always suspicious. Single mass submissions are always suspicious.

More on git

During the first tutorial, we will assist you to set up your git repository. Our git repository is open, in the sense that you can access it from outside the university. The idea is to have the working copy of your project on the git server, and 'clone' it to your local machine if you want to work on your code. Once you have made local changes, you can then commit it back to the git server. This serves as an excellent backup mechanism for you, and I insist that you commit your code on a daily basis to git. This also implies that I will not accept excuses for failures of your private computers, since you are supposed to use git to maintain backups. Also, this serves as a progress indicator, and can be used to dispute plagiarism claims.
Do not use different git branches; stay in the master branch.
Kindly note that it is against the plagiarism policy of the university to use publicly accessible repositories to post your code. Since you submit the work as part of a degree, the output belongs to the university – as a staff member, I am subjected to the same rules.
You MUST follow the directory structure convention, for the handin. If you do not do that, the marking script will not work. You should not, under any circumstances whatsoever, use hard coded file paths in your code.

Subdirectory structure on git

`src/` - source files, including Translate.java and any custom classes.
`bin/` - redirection folder for compiled .class files. The class files **must not** be commited to git, but the directory must exist.
`out/` - output directory for saved text files. These output files **must not** be commited to git, but the directory must exist. Your code should write any output to this directory.
`tests/` - test directory containing test cases and expected outputs, as well as the test script. This will be automatically provided through git for you.
In your root directory on git, you must have a `.gitignore` file, and a `README.txt` file. The `.gitignore` file is used to ignore certain files when submitting to git from your local machine. This includes, amongst others, `.class` files. Also make sure that no 'hidden' files appear in your git directory when you commit (typically, this would be `.swp` files, or system files. **If there are files are present in your git repository that should have been ignored, you will be penalised severely**.
The `README.txt` file must at least include your student number identification, and the title of the project. It must include instructions on how to run your code from the command line. It must include the names of any libraries required to run the project (such information is usually storied in a `requirements.txt` file, but for this project we include it in the `README.txt` file). You must also put in an acknowledgement of all other sources of information which was used to code the project, such as demis or other students with whom you discussed the project, or internet sources on which non-core parts of the project may be based. Remember that you have to include whether I gave my permission for this, with the accompanying date on which the permission was granted.

Handins

You must hand in your project on git. The dates and times are given in Section 0.1. This means that you have to be sure that your git is working at all times, and that you do not forget your access information. All handins must conform to the style conventions. Feedback will be provided for each handin, so that

you have time to fix any errors. Note that all the handins, if ignored, carry a penalty of -3% each, to be applied to your final project mark. That is, you will receive the penalty of -3% if you do not hand in, or if the requirements of the handin is not met. For example, if the git pipeline shows style errors, or the script fails on your handin, you would get a -3% penalty.
Note that the handins are dependent on each other – you must get each one correct, because the next handin depends on a working copy of the previous handins. There are **no remarks** of previous handins. Your mark for each handin stands as is. However, you have time to talk to the demis and find out why you may have lost marks, and repair that for the next handin. Please refer to the module framework for more detail.

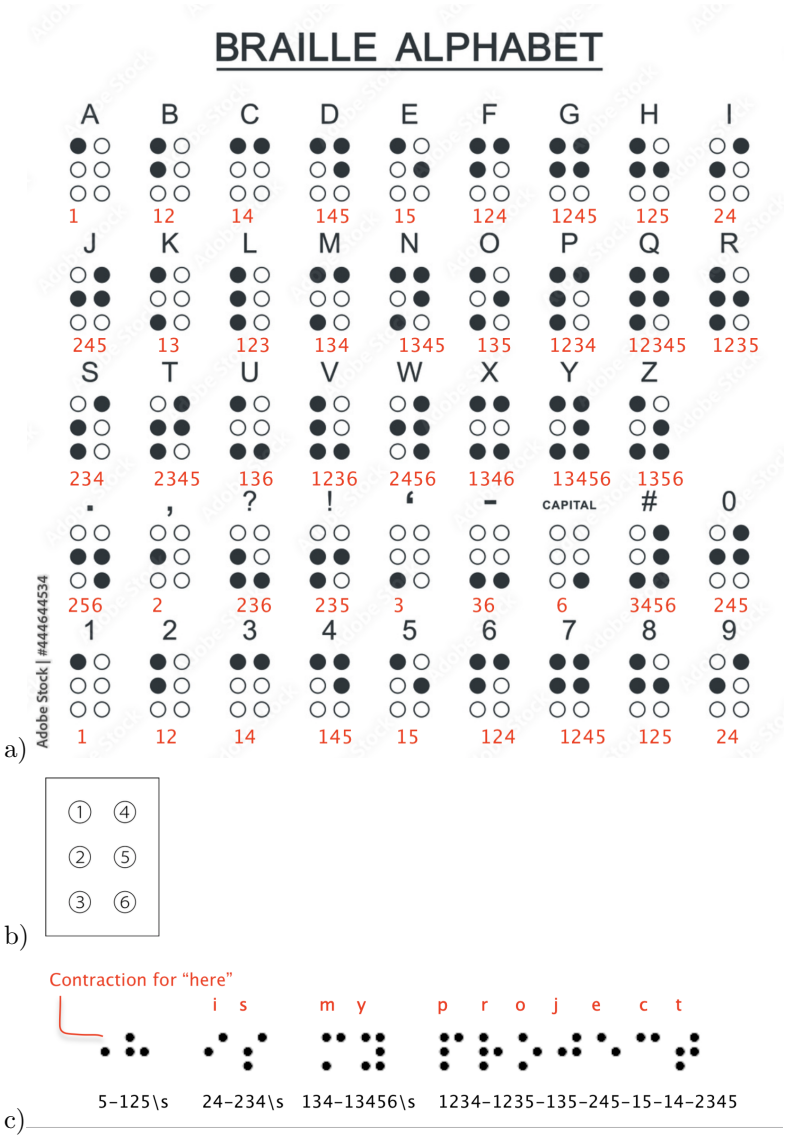

Figure 2: a) Braille characters for the alphabet and digits; b) the layout of Braille cells; c) Example Braille to brf translation

Specification

The project aims to translate Afrikaans text to Braille rich format (brf), and vice versa. You will be provided with a list of Afrikaans words. **It is not at all necessary to understand Afrikaans in order to do this project.** Initially, you will be expected to produce a Braille character brf code (see below) for each symbol in the input text (this is a to z). Then, you have to add control characters, such as characters that indicate that the next symbol must be a capital letter, or a number, or a punctuation mark. Lastly, you have to implement contractions, where commonly used morphemes or words are contracted to a single character, depending on a given sentence context.
Braille characters are represented as six dots in three rows of two dots; for convenience, you should use the numeric indices of each dot to represent a Braille character as a series of dots. The images below provide a summary of the Braille dots for the symbols a to z, and digits, the indices that indicate the Braille cells, and an example translation from text to Braille indices. These indices are known as `brf` codes. Note that a space is indicated by the symbols
`s` for clarity here, but in your output, it should simply be a space.
You will be provided with a word list on SUNLearn, and the rules for contractions are in the appendix to this document. The contractions are divided into four levels, from level 1 to level 4. Each level is assumed to also contain the contractions of all lower levels. So, level 3 contains all the level 1 and level

2 contractions as well. Each level also has some sublevels. You need to provide a runtime argument to indicate the level of your translation.
To start off, you must translate all alphabet symbols to and from Braille. That is, it is a one-to-one translation from text symbols to brf codes, and simply requires you to decide which lookup table Java object is most appropriate. Next, you should look at control characters – if you have for example a capital letter in your text, you must add a control character to the brf translation; similarly, if you read a control character in the brf, you must apply it to the text translation. Note that numbers require a control character. Levels 2, 3 and 4 have increasingly intricate rules to translate between text and Braille, where the context of a word or phrase determines the contractions to be used.

Testing your own work

To test your own work, the directories as given before must be present in your repository's main directory if you wish to run test cases yourself. Note that the `tests` directory is not provided directly with the git repository, but will be uploaded onto SUNLearn. You will need to download this directory and put it in your local repository. You must not upload the contents of this directory to the server repository.
The marking script `testscript.py` will be provided inside `tests/`. This script is the same script that will be used for the final evaluation, but only a small subset of all test cases will be provided for each mode. This script can be run manually on your local machine if you wish to check that your implementation will pass the provided test cases. This is in no way an accurate description of your final mark (since it is only a small subset of the test cases). However, it is valuable to confirm that your input and output formats are correct, and that you followed the specifications properly. Note that running the script on your local machine is a helpful tool, but not a guarantee of correctness. You must run the script on the git server for a more accurate indication of correctness.
Within a file called `cases.py` you can see the list of test cases if you wish to run them manually instead of using the script, or if you wish to add custom cases. To see how to run the testing script, see the `README.md` file in your root directory on the git server.

Output naming convention

Each phase will require saving your output to a text or brf file. See the table below for the output names required for each phase. The output file must have the same name as the input file, but with the correct file extension. This is essential, as it is required by the marking script, so not following this means we cannot mark your submission.
All files must be saved in the `/out/` directory.
Text files must be saved as `.txt`, and Braille files as `.brf`. Note that the level may range from 1.0 to 4.1. The direction indicates a translation from Braille to text (b2t) or text to Braille (t2b).

Direction	Level	Name extension
b2t	i.j	_b2t.txt
t2b	i.j	_t2b.brf

For example, on a test input file `case1.txt` for translation from text to Braille, your output file must be named `case1_t2b.brf`. And, on a test input file `case2.brf` for translation from Braille to text, your output file must be named `case2_b2t.txt`.

Style

You are expected to follow the Google Java style guide. Your project will be run through a style checker, which will contribute to part of the mark. This style checker will be run every time you commit to your repository. You will also be required to document your program using the JavaDoc style and format.

GUI

There is no GUI necessary for the git handins, as the testscript runs on terminal input and output. However, you may implement a basic GUI for demo purposes. This GUI must simply be able to show the input and output next to each other. Any other features are optional. No extra marks will be awarded for GUIs at all.

The first command-line argument will be used to distinguish between these two options. An argument of `noGUI` indicates no GUI, and will be used for marking with the test script. Ensure that no GUI output is produced in this case. An argument of `GUI` indicates that a GUI is present, and will be used in the demos. It is up to you to decide and describe the command line arguments and output methods of your program in this case.

Phase 1:

Phase 1 involves reading in a text file in Afrikaans, and producing the corresponding brf, or vice versa. The text file may include numbers. The control characters needed in this phase are only those for a single character letter, a word starting with a capital letter, a word in fullcaps, and numbers. See the appendix for more details.

Input

The program is run with

```
java src.Translate noGUI <direction> <level> <filename.xxx>
```

The first argument indicates that no GUI is to be used, the second indicating whether the translation is to be from text to Braille (`t2b`) or Braille to text (`b2t`), the third argument specifies a given level between 1.0 and 4.1, and the last argument is the filename of the file containing the text/brf to be translated.
The input file will either contain lines of text, with numbers and punctuation, or it will contain lines of Braille code in brf format. The brf format is simply lines with numbers, possibly separated by dashes, and spaces.
For example text and brf files, see the student test cases on SUNLearn.

Phase 2: Control characters, numbers, and elementary punctuation

Phase 2 involves reading in a text file in Afrikaans, and producing the corresponding brf, or vice versa. You must have a full implementation of uncontracted Braille (see the appendix). This includes normal text symbols, punctuation, control characters, and numbers.
The input and output formats correspond to those of Phase 1.
The input file will be a text or brf file.

Phase 3: Contraction levels 1 and 2

Phase 3 revisits the work done in phases 1 and 2, adding extra functionality.
You now have to implement the contraction rules, specifically for all the sublevels of levels 1 and 2. More about these rules can be found in the appendix.
The input and output formats correspond to those of Phase 1.
The input file will be a text or brf file.

Phase 4: Contractions levels 3 and 4

Phase 4 revisits the work done in phases 1 to 3, adding extra functionality. In particular, the last contraction rules have to be added.
The input and output formats correspond to those of Phase 1.
The input file will be a text or brf file.

Error handling

Some error handling will be required. That is, a few test cases will have invalid input parameters or input files and an error message must be sent to standard **error** (not standard output). The errors below are listed from highest priority to lowest priority – that is, if the GUI argument is an integer and not a string, then `invalid argument type` must be displayed, and not `invalid mode`.
Here is the list of errors that must be catered for, as well as their expected outputs:

INPUT ERRORS

1. **Invalid number of arguments**: If the incorrect number of arguments is given, then the following error must be displayed:

   ```
   Input Error - Invalid number of arguments
   ```

   This error message must be prioritised over any other error message.

2. **Invalid argument type**: ~~If an argument is of the wrong TYPE (for example, a GUI argument of "123" is given), then this error must be displayed:~~

   ```
   Input Error - Invalid argument type
   ```

3. **Invalid GUI**: If the GUI is not 'GUI' or 'noGUI', then this error must be displayed:

   ```
   Input Error - Invalid GUI argument
   ```

4. **Invalid direction**: If the direction is not 'b2t' or 't2b', then this error must be displayed:

   ```
   Input Error - Invalid direction
   ```

5. **Invalid level**: If the level is not in $\{1.0, 1.1, 1.2, 1.3, 2.1, 2.2, 2.3, 2.4, 3.1, 3.2, 3.3, 4.1\}$, then this error must be displayed:

   ```
   Input Error - Invalid level
   ```

6. **Invalid input file**: If the file name is given, but the file cannot be opened or does not exist, the following error must be displayed:

   ```
   Input Error - Invalid or missing file
   ```

Text to Braille or Braille to text ERRORS

7. **Invalid file content**: If a text file or brf file contains invalid symbols, the following error must be displayed:

   ```
   Input Error - Invalid file content
   ```

   An example of invalid symbols in a text file could be non-printable characters, and in a brf file could be digits in brf codes that fall outside the range 1 to 6.

Appendix A: Different levels of contraction rules

There are different types of contractions. If the type of contraction is `word`, then the contraction can only be applied on a whole word, and not if the characters form a prefix or suffix, or occur in the middle of the word. Consider the sentence `Asseblief, die as is nie drasties gesond soos kaas nie`. In the table below, we show on which word in the sentence the contraction will be applied for the symbols `as`:

Word	Prfword	Midword	Sufword	Always
as				
	Asseblief			
		drasties		
			kaas	
				Asseblief, as, drasties, kaas

There is also a more difficult contraction type not mentioned above. This is the so-called lower contractions, with type `lword`, which occur on level 3.2. These are more complicated to check, as they are affected by context and punctuation. The translation with lower word contractions are dependent on flags set by all other translation sections. If the correct conditions match, the contraction is handled in the same manner as the other contractions.
Each level has its own contractions, and these are now listed below, showing the type.

Level 0: Uncontracted

If numbers are present, those must be explicitly indicated: `numsign 3456`. Note that once this code is found, the subsequent symbols are treated as digits, until an alphabetic character or a space is found. This allows for the use of punctuation within a number, such as the decimal number `3.14`.
Capital letters can be applied only to the next letter, or to the whole next word:

```
capsletter 6
begcapsword 6-6
```

All the alphabet letters `a` to `z` are written simply as their equivalent brf codes (refer back to Figure 0.1). The brf codes for punctuation symbols are:

```
always , 2
always . 256
always ; 23
always : 25
always = 6-2356
always ! 235
always # 456-1456
always / 456-34
always ? 236
always % 46-356
always ^ 4-26
always ~ 4-35
always & 4-12346
always * 5-35
always ( 5-126
always ) 5-345
```

Diacritics (here shown with respect to the letter e with brf code 15) are written as:

```
è 45-16-15
é 45-34-15
ê 45-15
ë 56-15
```

Punctuations can occur before or after words. In the case of numbers, punctuation can be found before, in-between and after the number. Certain punctuation marks also have multiple braille codes depending on where they are encountered, for example quotation marks. The rules on punctuation are as follows:

- Punctuation may occur before and after numbers.
- Signs may occur before, between and after numbers.
- The only punctuation allowable in the middle of words is the hyphen.

The checking for punctuation is isolated from the alphabetic translation. As alphabetic translation is influenced by punctuation, the checks for punctuation occurs before alphabetic translation.
The contractions for the subsequent levels are listed below. Note that the uncontract symbol may come into play in any of the levels. For example, in Level 1.1, the word `baie` is contracted into a symbol `b`. So, for example, if translating text on level 1.1, and something like "Section B" is to be translated, the `B` must have the uncontract symbol in front of it (otherwise it would be translated in "Section baie"). So, the brf code 56 is used to show that a contraction should not be applied: `nocontractsign 56`.

Level 1.1

```
word as 1
word baie 12
word sal 14
word dan 145
word ek 15
word om 124
word geen 1245
word hy 125
word is 24
word jy 245
word kan 13
word liefde 123
word my 134
word nie 1345
word ook 135
word plek 1234
word ons 12345
word reeds 1235
word so 234
word tot 2345
word van 1236
word wil 2456
word het 1346
word sy 13456
word 'n 1356
```

A problem will now appear in backtranslations from Braille to text. For example, if one reads the code for Z (1356), how do you know whether it means `z` or `'n`?

Level 1.2

```
always of 12356
always die 2346
always met 23456
always sk 16
always aan 126
sufword al 1246
always ou 1256
always was 346
always ui 345
```

Level 1.3

```
always dag 5-145
always familie 5-124
always goeie 5-1245
always hul 5-125
always jul 5-245
always kom 5-13
always lewe 5-123
always moet 5-134
always nooit 5-1345
always onder 5-135
always party 5-1234
always reg 5-1235
always sien 5-234
always tyd 5-2345
always uur 5-136
always vol 5-1236
always werk 5-2456
always kry 5-13456
always nog 5-1356
always dieselfde 5-2346
word hulle 45-125
word julle 45-245
```

Level 2.1

Note that we now encounter the same abbreviation for two different cases. For example, both `ander` and `an` has the same abbreviation. This will complicate your Braille to text translations.

```
word ander 12346
word want 123456
word eindelik 146
word iets 1456
word een 156
word erken 12456
word wat 246
word stil 34
always an 12346
sufword ge 123456
always ei 146
always ie 1456
always oe 246
always ee 156
always er 12456
always st 34
```

Level 2.2

```
always doen 45-145
always seker 45-234
always weer 45-2456
always antwoord 45-12346
always wanneer 45-123456
always iemand 45-1456
```

Level 2.3

```
always heid 46-145
```

```
always tjie 46-15
always agtig 46-1245
always djie 46-1456
always lik 56-13
```

Level 2.4

```
word brief 12-1235
word braille 12-1235-123
word duidelik 145-13
word daarom 145-134
word daarna 145-1345
word daarop 145-1234
word daar 145-1235
word daarvan 145-1236
word dikwels 145-2456
word daardie 145-2346
word goed 1245-145
word groot 1245-1235-2345
word grootste 1245-1235-2345-34-1235
word heeltemal 125-123-2345
word hom 125-134
word haar 125-1235
word hoewel 125-2456
word hierdie 125-2346
word jaar 245-1235
word kind 13-145
word kinders 13-145-234
word kort 13-2345
word laat 123-2345
word maak 134-13
word maar 134-1235
word moontlik 134-2345-13
word miskien 134-16
word omtrent 135-2345
word omstandighede 135-34-145-15
word probeer 1234-12
word paar 1234-1235
word soos 234-234
word teen 2345-1345
word terwyl 2345-2456
word vandag 1236-145
word word 2456-145
word waarom 2456-134
word waarop 2456-1234
word waar 2456-1235
word wees 2456-234
word weet 2456-2345
word waarvan 2456-1236
word gebruik 123456-12
word gedurende 123456-145
word gemaak 123456-134-13
word genoeg 123456-1345
word omdat 134-145
word gewees 123456-2456-234
word geweet 123456-2456-2345
word skryf 16-1235
word alhoewel 1246-125-2456
```

```
word geword 123456-2456-145
word alles 1246-123
word almal 1246-134
word alreeds 1246-1235
```

Level 3.1

```
always graag 45-1245
always net 45-1345
always voor 456-1236
always self 45-234
always tussen 45-2345
always vir 45-1236
always woord 45-2456
always dit 45-2346
always darem 456-145
always hard 456-125
always koning 456-13
always mens 456-134
always nuwe 456-1345
always oor 456-135
always praat 456-1234
always tog 456-2345
always dat 456-2346
```

Level 3.2

```
midword aa 2
sufword be 23
midword be 23
sufword on 25
midword on 25
lword deur 256
sufword deur 256
lword en 26
sufword en 26
midword en 26
prfword en 26
midword ge 123456
lword te 235
sufword te 235
midword te 235
midword oo 2356
lword se 2356
lword hier 236
lword in 35
sufword in 35
midword in 35
prfword in 35
lword by 356
sufword by 356
midword by 356
```

Level 3.3

```
word daarin 145-1235-35
word daarvoor 145-456-1236
word dadelik 145-56-13
word sodat 234-456-2346
```

```
word totdat 2345-456-2346
word waarin 2456-1235-35
word waarvoor 2456-456-1236
word waarheid 2456-1235-46-145
word begin 23-1245
word behoort 23-125
word besluit 23-234-123
word ontvang 25-2345-1236
```

Level 4.1

```
always ig 12345
always el 3456
always ver 36
```