Manual for CsPTools v0.3.1

Thomas McFadden
DFG Project: Auxiliary selection in the history of English
Universität Stuttgart

July 26, 2005

Abstract

CsPTools is a package of tools written in perl which assist in the processing and interpretation of files output by the CorpusSearch program (henceforth CS). Much of the functionality centers around the use of coding with CS, which typically involves extensive human editing of CS outputs, and thus implies a great deal of time expenditure and an associated high rate of errors. CsPTools has been designed to automate parts of this process, reducing the amount of human time necessary and addressing specific issues of error avoidance and detection. Version 0.3.0 was the first public release, version 0.3.1 adds a new program `editcode`, but is otherwise essentially a bug-fix, and is still emphatically in the beta stage.

Contents

1 General package info

1.1 Downloading

CsPTools can be downloaded from the auxiliary selection project site:

`http://ifla.uni-stuttgart.de/~tom/project/CsPTools.shtml`

See Section 1.3 for license information.

1.2 Installation, configuration and requirements

CsPTools obviously requires perl. It will definitely work with perl v5.8 and later, and seems to work in a quick test on v5.6.x as well, though this may not hold true under extensive use. Due to the use of certain pragmatic modules, it will almost certainly not work on anything older than that.

As noted above, CsPTools works with files output by the CorpusSearch program. It is however maintained and distributed independently. CS is now open source, and is available from the following SourceForge page:

`http://corpussearch.sourceforge.net/`

Since most of the scripts only really deal with CS output, they can actually be used on a system where CS is not installed (on files that have been moved from a machine where CS **is** installed). The only exception to this is the `autocs` script which does directly call CS, being really just a front end for it. CS v2 differs from v1 in a number of non-trivial respects relevant to the tools here.[1] They are intended to be set up to work with (the output of) CS v2. However, I have attempted to maintain compatibility with v1 output, and since early versions of the scripts were developed with CS v1, there may still be things lurking in dark corners that are incompatible with v2. I am especially interested in correcting such bugs, so let me know if you find any.

The `analyzer`, `next` and `editcode` scripts depend on emacs. They don't require a particularly recent version. The way they are set up now, I think they should work without special configuration, but if you want a specific version of emacs to be run (rather than the default in your PATH) you can play with the `system` lines that actually call emacs. In the same way you may also be able to get them to use a different editor, though for `analyzer` and `editcode` you'd have to figure out how to get the editor to open at a specific line number in the file.

The `next` script depends on the `codefinder` script. I'm curious whether the way I have it set up will work for people on other machines. I think it should, as long as you put `codefinder` in a directory that's in your PATH. I may change the way this works in future versions to eliminate the dependency.

The current version should work on all UNIX-like systems (UNIX, Linux, MacOSX, the BSDs...). It may work on other operating systems, but I would be surprised if it did so without modification. I'll try to fix it up to do so in later versions if there's interest.

Installation should be pretty straightforward. The package consists of ten perl scripts and a module file, `CsPTools.pm`, on which they all depend. If you already have a directory where you put perl module files, just put `CsPTools.pm` there. If you don't have such a directory or don't know what I'm talking about, then create such a directory, perhaps something like `/usr/local/lib/perl` or `/home/tom/perllibs`. If you want other people on the machine to be able to use what you install, and you have sufficient permissions, go for something like the former. If this is just for your own use, or you're working on a machine where you don't have administrative permissions, go for something like the latter. The important thing is that, no matter what you choose, you then need to tell perl to look in that directory. To do this you need to modify (or set) the environment variable PERL5LIB in your shell initialization file. If you're using csh or tcsh, add this line to your `.cshrc` file (substitute the directory you want to use for `/usr/local/lib/perl`):

```
setenv PERL5LIB "/usr/local/lib/perl"
```

[1] E.g., `CODING` strings are now inserted into a different position in the tree.

If you're using sh, bash, ksh, or zsh, add the following to the relevant initialization file:

```
PERL5LIB=/usr/local/lib/perl; export PERL5LIB
```

One other environment variable will have to be set to ensure that you can use the `autocs` program. It runs CS for you, and since different people run CS in different ways, based on their java set up, `autocs` needs a little help. So in your shell initialization file, set the `CS` environment variable to the command you use to run CS. Note: if you use an alias when you actually run CS from the command line, make sure you set the environment variable to whatever the alias is set to, not to the name of the alias! The line in my .cshrc file looks like this:

```
setenv CS "java -classpath /usr/local/java/latest_flagCS csearch/CorpusSearch"
```

The ten perl scripts need to be put into a directory that's in your PATH. You should also double-check to make sure that their execute permissions are set correctly. They assume that perl itself (or a link to it) is in `/usr/bin/perl`. This is fairly standard, but if your machine has perl somewhere else, you'll need to change the first line of each script appropriately. Note that on some machines, `/usr/bin/perl` exists but does not contain the most recent version of perl. E.g., on the UPenn linguistics department's server babel, the most recent version is currently in `/pkg/bin/perl5.8.0`, whereas the version in `/usr/bin/perl` is 5.005_03, which will not work with CsPTools.[2] So on that machine you'll have to change the first line of each script to:

```
#!/pkg/bin/perl5.8.0
```

Hopefully, future versions of the CsPTools package will follow standard practice for perl modules and come with configuration and installation scripts to simplify the process described above.

1.3 License

1.4 Package-wide conventions

CsPTools is not a single program, but a collection of 10 scripts performing different functions which simplify the use of CS and the interpretation of its output. The scripts are strictly speaking independent of one another (with the exception of `next`, which depends on `codefinder`), but they all depend on `CsPTools.pm`, which contains code shared among the scripts. I have attempted to maintain a certain amount of consistency in the behavior of the scripts.

[2] For those of you working on babel: double check this when you're setting things up. Rumor has it that things might be set up in a more normal fashion with the most recent perl in `/usr/bin` in the future, possibly by the time you read this.

All of the scripts are command-line tools following fairly standard UNIX/Linux conventions for command-line options. They all understand `--version` and `--help`, which reports a brief description and usage information for the relevant script (including all command-line options). Most options can be specified either in long form (`--help`) or the corresponding traditional short form (`-h`).[3] When this is not the case, it will be indicated in the output of `--help`. As of v0.3.1, all scripts consistently recognize `-h` and `-v` as abbreviations for `--help` and `--version` respectively. `-V` is consistently used as the short form of `--verbose` for those scripts that use that option. Short options can be bundled in the usual way, i.e. `-mot` is the same as `-m -o -t`. Note that this means that the long versions **must** be preceded by two dashes. `-xy` will be interpreted as `-x -y`, or potentially as `-x` taking the argument `y`, not as `--xy`. Errors of this kind can lead to unexpected and difficult to intepret output. When fed a non-existent option or an option with the wrong kind of argument, the scripts will thus abort with an error statement rather than trying to run with potentially bad results. I have attempted to consistently use the same name for options that are valid in multiple scripts. All of the scripts for which it makes sense default to sending their output to STDOUT (the terminal),[4] though this output can of course always be redirected to a file with standard shell tools.

One of the benefits of using perl for this package is that it makes it very easy to include support for regular expressions, which are not available in CS. Wherever I thought it made sense, I have set up the scripts to allow the user to specify input in terms of regular expressions. The class of regexes accepted are of course those used by perl, which differ somewhat from those used by grep, sed and awk (in the direction of being more powerful). If you're not familiar with them, there is extensive documentation in the perldoc system. If perldoc is properly configured on your system, type `perldoc perlre` for an exhaustive reference, `perldoc perlrequick` for a quick overview or `perldoc perlretut` for a tutorial. If that doesn't work, try typing `man` instead of `perldoc`. I have tried to be consistent in the way that user-supplied regexes are handled by the various scripts. As with grep (and perl), a regex is considered to match the string in question when it matches with any substring contained in the string. It does not have to match the entire thing. So a regex supplied as `'trans'` will match `'trans'`, `'intrans'`, `'%trans'`, `'transit'` and so forth. If you want an exact match of the entire string, i.e. if you want just `'trans'` and not the others to be counted as a match, use the appropriate anchor characters. Here you would want `'^trans$'`. Note that if your regex contains any fancy characters like the anchors, you'll have to quote it so that they get past the shell. In general, single-quotes, as in the examples just given, will do the right thing on UNIX-like systems. Please report any bugs or inconsistencies you might come across in the handling of regexes. They can be a bit dicey. I also welcome suggestions of other places to allow their use in the user input.

1.5 Reporting bugs and such

As noted above, CsPTools is very much still a beta. Most of the scripts have been used extensively within the project for which they were created and have already seen quite a bit of improvement, but they haven't seen much use yet elsewhere, and I have no idea what's going to happen when other people on other systems try to use them.

Please report any bugs, inaccuracies, typos, suggestions for improvement, etc. to the following email address:

`tom@ifla.uni-stuttgart.de`

[3] The short versions of the options didn't actually work with a lot of the scripts in v0.3.0. This bug has been fixed in 0.3.1.

[4] This obviously does not make sense (and thus is not the case) for scripts which are meant to produce files, like `autocs`.

I am very much interested in getting this stuff to work, so I will respond and do my best to fix things. I am particularly interested in whether and to what extent the tools work on systems set up differently from mine, since I haven't been able to do much testing. I am also interested in suggestions for improvements, additional features, whatever. And you are of course welcome to try to fix things yourself if you know perl. Let me know what you come up with. Most of the code is heavily commented, so there's at least a decent chance you'll be able to figure out what's going on.[5]

2 The programs

The subsection on each program begins with what is output when the `--help` option is used (printed in `typewriter` typeface), giving a quick synopsis, usage information and the full list of command-line options. This output is then followed by lengthier discussion of how the program is used, along with examples, and a note on any changes from earlier versions.

2.1 `analyzer v0.1.2`

```
        Analyze (multicolumn) codes in a CorpusSearch output file.

Usage:
        analyzer [options] file

Command-line options:
-1, ... -9       <string> Restrict to codes with <string> as column 1, ... 9
-e, --edit                Edit sentences matching query in emacs
-h, --help                Print this helpful message
                            in column <#>
-q, --quit                Print results and quit.  Don't prompt for editing.
    --rank       <#>      List values for column <#> with number of matches
                            for each, sorted by number
-s, --sort       <#>      List values for column <#> with number of matches
                            for each, sorted by value
-r, --regex               Treat <string> following -1 ... -9 as a
                            regular expression
-v, --version             Print version information
-x, --xy         <a:b>    Print as table, with values from column <a> on
                            x-axis and values from column <b> on y axis
```

Change in v0.1.2: `-r` is now short for `--regex`. In v0.1.1 and previous is was short for `--rank`.

`analyzer` processes and reports on the coding strings in a CS output file. It is especially designed to deal with multicolumn codes and has several options for filtering and formatting the output. When run without any options, it produces an output like this:

```
1804 codes matched your query:
    # line no. code
    1       32 ncomp:pres:h:live:m2
    2       47 intrans:cond:h:be:m2
    3       67 intrans:plu:h:set:m2
```

[5] Beware – some of the comments are out-of-date, and I'm not an experienced programmer, so what you find might be a bit ugly.

```
4       85 intrans:pres:b:come:m2
5      100 intrans:plu:h:be:m2
6      116 intrans:plu:h:be:m2
.
.
.
```

The file it was run on here contains all the intransitive perfect clauses from the PPCME2, coded for type of intransitive, tense/mood, auxiliary, main verb and period. The top line indicates how many coding strings were found. The codes found are then listed, one per line. The first column numbers the hits sequentially so that they can be referred to later. The second column contains the line number where the coding string was found. The third column contains the coding string itself. The user is then presented with a prompt:

```
To view a hit in emacs, enter its number, or q to quit.
>
```

To view the position in the file where a particular coding string occurs (e.g. to read the sentence or edit the coding string), the user can then enter the corresponding number from the first column in the output. To get back to the `analyzer` prompt, just exit emacs normally. If the user knows ahead of time that she wants to view all of the hits in this way, she can use the `--edit` option. Then she can go through the hits in emacs sequentially by hitting enter at the prompt. The `--quit` option causes `analyzer` to quit immediately after printing the results, skipping the prompt.

There is a series of options for restricting the output to coding strings with specific values for one or more columns. The basic ones are simply numbers corresponding to the columns themselves. So to restrict the preceding output to codes with `plu` in column 2, use `analyzer -2 plu`. Multiple columns can of course be referred to simultaneously, so in the example file being dealt with here, `all.out`, we can get all coding strings involving counterfactual perfects of *come* with auxiliary HAVE in the 3rd ME period with `analyzer -2 ctf -3 h -4 come -5 m3 all.out`, yielding the following:

```
5 codes matched your query:
    # line no. code
    1     5891 intrans:ctf:h:come:m3
    2    11075 intrans:ctf:h:come:m3
    3    21664 intrans:ctf:h:come:m3
    4    31872 intrans:ctf:h:come:m3
    5    31887 intrans:ctf:h:come:m3
```

By default, the string following a column number option is treated literally, and only those coding strings are reported which contain exactly that string in the relevant column. With the `--regex` option, however, one can tell `analyzer` to treat the string as a regular expression, and search for all hits where the regular expression matches somewhere in the relevant column.

There are two options for generating reports on what values occur in a given column. Both, when followed by a column number, will cause `analyzer` to print out solely the distinct values that appear in that column, along with the number of times that value occurs in the file. They differ only in the order. `--rank` puts them in order of frequency, with most frequent first, while `--sort` puts them in alphabetical order by value. So to find out which verbs occur with auxiliary BE in the second ME period, in order of frequency, we could use `analyzer -3 b -5 m2 --rank 4 all.out`:[6]

[6]Note that the `-3` and `-5` option are still used here to narrow down the coding strings we're interested in, and `--rank` operates on that smaller set. It is generally the case with the column number options that you can use them to narrow down the input to one of the reporting options.

```
The following appeared in column 4 in codes matching your query:
come:              11
wend:               4
rise:               2
go:                 2
spring:             2
run:                2
.
.
.
```

...or for alphabetical order, `analyzer -3 b -5 m2 --sort 4 all.out`:

```
Here are the codes that fit your query:
become:        1
come:         11
escape:        1
fall:          1
flee:          1
go:            2
.
.
.
```

`analyzer` can also draw simple tables to view the interactions between the values in two columns. For this use the `--xy` option, followed by two numbers, separated by a colon, where the first number is the column you want displayed on the x-axis, the second the column you want on the y-axis. So to see the frequency of HAVE vs. BE with *come* over the four periods of ME, we can use `analyzer -4 come --xy 5:3 all.out`:

```
Here's your grid:
               m1      m2      m3      m4
b              64      11      97      75
h              1       0       14      11
```

As of v0.1.2, `analyzer` now deals reasonably well with multiple input files. If you run it without any special reporting options (i.e. `--rank`, `--sort` and `--xy`), it will list the codes with the name of the file they were found in, and will correctly determine linenumbers in multiple files. This also means that editing works correctly with multiple files. `analyzer` does not yet allow separate treatment of multiple files with the special reporting options, i.e. listings with `--rank` or `--sort` and grids with `--xy` will just report on the totals for all files listed. Normally, this is what you want anyway. If I can think of a good reason to implement fancier treatment of multiple files (like say grids where one axis has filenames), I may do so in later versions.

2.2 autocs v0.2.1

```
        Run CorpusSearch with strict file-naming conventions, producing
        a separate output file for each input file.

Usage:
        autocs [options] [query file] file(s)
```

```
Command-line options:
-a, --auto                Use "out" as output directory.
-d, --dir        <dir>    Use <dir> as output directory (overrides -a).
-h, --help                Print this helpful message.
-i, --id         <string> Set query id to <string>
-m, --manual              Don't automatically determine query id.
-o, --overwrite           Overwrite older output files without asking.
-q, --query      <file>   Use <file> as query file.
-s, --suffix     <string> Use <string> as extension on output files.
-t, --test                Don't actually run CorpusSearch, just report what
                            would have been run
-v, --version             Print version info.
```

`autocs` is a front-end for CS which automates repetitive parts of the process to avoid errors and save time. One main feature is that it allows you to produce a separate output file for each input file that a particular query is run on. If you run CS on several files at once, say all the files that make up a given corpus, it will give you a single output file containing hits from all the input files. Often this is what you want, and CS does a very intelligent job of reporting statistics for the individual files that went into making a larger output file. But sometimes this is not what you want. Sometimes you want a separate output file for each input file (e.g. to provide a simple way to chunk subsequent editing work, or if you want to be able to handle files from different periods separately without writing an additional query to split them up). `autocs` provides this possibility by calling CS individually for each input file. This could be done by hand, but would be slow, tedious and prone to error.[7]

The other main feature of `autocs` – which is especially important when you're creating lots of files, as you will when producing a separate output for each input – is that it facilitates the use of a strict file-naming convention. The convention is simple. The output file will consist of the input file, minus the extension (`.psd` or whatever), followed by `_string`, where `string` is an identifier associated with the query being run, followed by the appropriate extension, either `.out` or `.cod`. A simple way to make this work is to prefix the name of each query with a unique number, and to use that number as the query identifier. The nice thing about this system is that each search run on a file will suffix an additional query identifier, meaning that, given an output filename, you can determine what the original corpus file was, which queries have been run on it, and in which order.

E.g., let's say we have three query files, `1vbn.q` which finds clauses containing a perfect participle, `2come.q` which finds clauses containing a form of the verb *come*, and `3hb.c` which codes clauses according to whether they contain auxiliary HAVE or BE. Using our naming conventions, if we run these three queries in sequence on the corpus file `cmwycser.m3.psd`, we would have output files named `cmwycser.m3_1.psd`, `cmwycser.m3_1_2.psd` and `cmwycser.m3_1_2_3.psd`. If we came back several weeks later, and found the third file, we would know immediately – without having to examine the file itself – which queries were run and in which order.

`autocs` needs the following pieces of information to run: a query file, at least one input file, the query id, the extension for the output file(s), and the directory in which to put the output file(s). This info can be specified with command-line options and command-line arguments, and `autocs` can determine quite a bit automatically. The user will be prompted to supply anything that is not specified or inferable. The different pieces of information are determined as follows:

query file `autocs` first checks for the `--query` option. If this is not set, it checks to see whether the

[7] The ability to run searches separately on multiple files may actually appear in future versions of CS.

first argument ends in `.q` or `.c`. If neither of these supplies a query file, the user is prompted to supply one.

input file(s) After an initial argument in `.q` or `.c` is stripped off, all remaining command-line arguments are treated as input files. If none are given, `autocs` quits with an error.

query id `autocs` first checks to see if a qid was supplied with the `--id` option. If not, it tries to determine the qid from the name of the query file. If the query file starts with a number, it takes that as the qid. If this doesn't work, then it prompts the user. Also, the user can specify with the `--manual` option that she wants to be prompted even if the name of the query file starts with a number.

output extension `autocs` first checks to see if an output extension was specified with the `--suffix` option. If not, it chooses `.out` or `.cod` depending on whether the query file ends in `.q` or `.c`. Again, the `--manual` option overrides this automatic determination and causes `autocs` to prompt the user for an output suffix.

output directory `autocs` first checks to see if an output directory was supplied with the `--dir` option. If not, then if the `--auto` flag is set, it sets the output directory to `./out`. If neither of these options is used, it prompts the user.

There are two additional flags which control how `autocs` is run. Since the program potentially does quite a bit automatically, involving lots of files and running for a long time, one might want to first do a dry run to make sure that everything is set up right. This can be done with the `--test` flag, which causes `autocs` to just print out what it would do, i.e. the series of queries that it would run, specifying the input and output files, but does not actually run CS. So if we do `autocs -t -d out/2 2verbs.c out/1/*.out`, where `out/1/` contains the output of a search run on all the files in the YCOE, we get an output that starts like this:

```
Query file: 2verbs.c
Corpus file: out/1/coadrian.o34_1.out
Output file: out/2/coadrian.o34_1_2.cod
------------
Query file: 2verbs.c
Corpus file: out/1/coaelhom.o3_1.out
Output file: out/2/coaelhom.o3_1_2.cod
------------
Query file: 2verbs.c
Corpus file: out/1/coaelive.o3_1.out
Output file: out/2/coaelive.o3_1_2.cod
------------
Query file: 2verbs.c
Corpus file: out/1/coalcuin_1.out
Output file: out/2/coalcuin_1_2.cod
------------
```

When doing searches with CS, one often has to run the same search several times while getting the query just right. This means overwriting outputs from previous searches. Dealing with this isn't so bad if each run yields a single output file, but if you're getting a separate output for each input, this can be a bit problematic. `autocs` checks for potential overwrites and prompts the user for how to deal with them giving four fairly obvious options:

```
File out/2/coaelhom.o3_1_2.cod already exists.
Choose one of the following options:
        [s]pecify different file name
        [q]uit
        [o]verwrite out/2/coaelhom.o3_1_2.cod
        overwrite [a]ll
```

The last option, `overwrite [a]ll`, means overwrite the current file, and if the same search is to be run on any further files, overwrite their old outputs as well, without prompting. If the user knows ahead of time that this situation is going to arise and wants to overwrite everything, she can set the `--overwrite` flag, which essentially choose the `overwrite [a]ll` option from the above dialog without prompting the user.

2.3 codefinder v0.5.3

```
        Find (single-column) coding strings in CorpusSearch output files,
        classify and report on them.

Usage:
        codefinder [options] file(s)

Command-line options:
-c, --codesfile  <file>   Use <file> as list of known codes
-h, --help                Print this helpful message
-i, --individual          Report on multiple files individually
-k, --known               Detailed report on known codes
-m, --machine             Detailed report on machine codes
-q, --question            Detailed report on question codes
-s, --string     <regex>  Detailed report on code strings matching <regex>
-u, --unknown             Detailed report on unknown codes
-V, --verbose             Detailed report on all codes
-v, --version             Print version information
```

Change in v0.5.3: In order to be consistent with the other scripts, `-v` is now short for `--version`, while `-V` is the abbreviation for `--verbose`.

What `codefinder` does is similar to what `analyzer` does – both report on the coding strings found in CS output files. However, the intended use of the two programs is rather different, and thus so are the details of their functionality. `analyzer` is meant to be used to learn about what sort of clauses are contained in an output file, to obtain numbers for doing statistical analysis of the data and so forth. It is for reporting results in the later stages of research using a corpus. On the other hand, `codefinder` is for earlier stages, to help in the actual hand coding of the examples prior to any statistical analyses. A lot of the coding of sentences can of course be handled by CS automatically, but anything not represented in the annotation structure must be coded by hand. This is a time-consuming and error-prone process, and `codefinder` is one of a series of scripts in the CsPTools package that automate this process as much as possible, cutting down on the time needed and reducing the possibility of errors.

Hand-coding generally consists in checking over and potentially correcting the codes assigned to clauses by CS.[8] If you're working on a non-trivial number of clauses, it's extremely helpful to be able

[8]Even in a case where it's not feasible to have CS guess on the proper coding string, it's still necessary to run a coding query inserting dummy codes before assigning codes by hand. This is the simplest way to insert the coding

to distinguish codes that have been checked out from those that have not, whether the codes had to be changed or CS got it right. Additionally, since files that have been coded by hand are typically fed back into CS for further processing, it's crucial to be able to catch typos that might have been made in the process. Finally, it often happens that a clause comes up whose classification is not entirely clear, and the researcher might like to mark it in such a way to allow her to come to it later. `codefinder` addresses these issues by classifying the coding strings it finds into four categories:

Machine codes
: Codes that have been assigned by CS and not yet hand-checked are distinguished by giving them a special mark. In the conventions used by our project, this is done by having CS coding searches only assign codes that start with the `%` character. Since that character does not otherwise occur in the historical English corpora series, it is an unambiguous identifier of a coding string assigned by CS. When a researcher hand-checks examples, she simply deletes the `%`, along with making any necessary changes. This makes it simple to track progress, and also speeds up the hand-coding process, because the researcher can move quickly from example-to-example by searching for the next `%`. `codefinder` classes every coding string that contains a `%` as a **Machine code**.

Question codes
: A similar strategy can be followed for instances where it's not clear what code should be assigned. A natural way to do this is for the researcher to include a question mark in the code she assigns, marking it as uncertain, to be double-checked, whatever. `codefinder` thus classes every coding string that contains a `?` character as a **Question code**.

Known codes
: In most cases, the actual coding strings that can be assigned will constitute a finite set. E.g., when coding perfect clauses according to the auxiliary used, there are only three possibilities: `have`, `be` and `null`. We can take advantage of this to search for potential typos and other errors: any coding string not on the list must be an error. To this end, `codefinder` can optionally be given a list of coding strings. Every coding string in the input file(s) that appears on this list will be classed as a **Known code**.

Unknown codes
: Any coding string that doesn't fit into one of the three above categories will be classed by `codefinder` as an **Unknown code**. This is of course where we look for errors.

The list of codes to be used to determine which strings are Known codes must be contained in a file with one coding string per line, i.e. something like this:

```
trans
intrans
psv
cogn
refl
ecm
predo
ob2
err
psv/intrans
clobj
```

string structure into the trees.

The name and location of such a file can be specified with the `--codesfile` option on the command-line. If no file is specified, `codefinder` automatically looks for a file named `codes` in the current directory and uses that. If no such file exists and no file was specified, `codefinder` operates without a code list, and all coding strings which are neither Machine codes nor Question codes will be classified as Unknown codes.

The default output of `codefinder` – for a file that is partially hand-corrected but has some typos and a couple questions – looks like this:

```
######################
Machine codes:
        %clobj             2
        %intrans           1
        %predo             1
        %psv              11
        %trans             1

        Total Machine     16
Unknown codes:
        pvs                1
        tarns              1

        Total Unknown      2
Known codes:
        clobj              1
        intrans            1
        predo              1
        psv                3
        trans              1

        Total Known        7
Question codes:
        ?intrans           1
        ?trans             1

        Total Question     2
######################
Grand total               27
######################
```

By default, when given multiple files to report on, `codefinder` will simply add everything together, every number reported being a total for all files mentioned. The `--individual` option tells it to report on each file individually. It will still give a grand total of all the coding strings found. Note that `codefinder` can be used in this way to get a very quick, but reliable count of how many hits are contained in a file or set of files.

Obviously, if your goal is to find and correct specific errors, the information printed by default may not be enough. It will tell you that you have errors and what they look like, but not where they are. With the `--verbose` option, you can tell `codefinder` to report the line numbers on which the particular coding strings appear. If you're only interested in the details on a particular class of codes – say if you're trying to track down the remaining examples that haven't been hand-coded –

you can use one of the corresponding options: `--machine`, `--question`, `--known` and `--unknown`. The output with the `--machine` option run on the same file as above looks like this:

```
####################
The following Machine codes were found:

File    Code                          Linenumber(s)
colacnu.o23_1_2_oth_3.cod
        %clobj                        287 419
        %intrans                      181
        %predo                        212
        %psv                          239 251 267 310 323 339 356 369
                                      384 400 438
        %trans                        196

Total Machine codes: 16
####################
```

If you're looking for information on a particular coding string or class of coding strings, you can use the `--string` option, followed by a regular expression. `codefinder` will then give a detailed report on every code that contains a string matching that regular expression. So if we run `codefinder --string 'trans'` on the file we've been using, we get the following output:[9]

```
####################
The following codes matched regular expression /trans/:

File    Code                          Linenumber(s)
colacnu.o23_1_2_oth_3.cod
        %intrans                      181
        %trans                        196
        ?intrans                      168
        ?trans                        139
        intrans                       89
        trans                         63

Total codes matching regex /trans/: 6
####################
```

I intend to incorporate the editing functionality of `analyzer` into future versions of `codefinder`. For the time being, you have to open the files in emacs and move to the examples yourself.

2.4 next v0.1.3

```
        Determine next file to be hand-edited in current directory,
        create copy with appropriate name and open in emacs.

Usage:
        next [options]
```

[9] If you want to search for codes that match a string exactly rather than containing it, e.g. if you just want **trans** and not **intrans** etc., you'll have to use anchor characters in your regex. Here, e.g., you could use **--string '^trans$'**.

```
Command-line options:
-c, --codesfile  <file>   Use <file> as list of known codes
-h, --help                Print this helpful message
-l, --last                Open last unedited file, i.e. follow inverse
                            alphabetical order
-n, --no                  Don't delete intermediate backup files,
                            and don't ask
-v, --version             Print version information
-y, --yes                 Delete intermediate backup files without asking
```

The `next` script is highly specialized and is only useful under very specific circumstances. It is for the case when one is hand-coding a large number of files in a single directory – essentially when one has used `autocs` to run searches creating separate output files for each input file in the corpus. Furthermore, it depends on the use of a particular file-naming convention: when hand-coding a file that was the output of a CS coding query, you make a copy and suffix `_h` onto the end.[10]

The script as currently written is of no use except under these conditions. But when these conditions are met, it can save quite a bit of time and avoid some rather tricky errors. The issue is this: the creation of separate output files for each input and the use of the file-naming conventions discussed above are well-motivated, but they have as a result that the researcher ends up with an explosion of files with long names like `cmwycser.m3_1_3.out_h` that must be typed correctly over and over again. The process of copying each file and opening it in an editor is thoroughly uninteresting, yet requires attention to detail. It's a recipe for errors, and `next` is way around the whole thing.

When run, it scans the current directory for files ending in `.out` or `.cod` that don't have a corresponding file in `_h`. It takes the first one ASCIIbetically, makes a copy with the `_h` suffix and opens it in emacs. The user can then hand-code it, and when she exits emacs, `next` takes over again. It first runs `codefinder` on the file so that the user can see right away if she's missed any machine codes or made any typos. It then gives the following prompt:

```
What would you like to do now?
        [r]eopen colacnu.o23_1_2_oth_3.cod_h
        [c]ontinue to next file
        [q]uit
```

This gives the user the chance to go back and fix up problems that `codefinder` has revealed, to move on to the next file that needs to be hand-coded, or to quit. If either of the latter two options are chosen, the user sees the following prompt:

```
Would you like to delete the backup file
        colacnu.o23_1_2_oth_3.cod_h~
first? [y/n] y
```

The named file is the backup that emacs creates automatically when you edit a file. Note that in the situation here, this will actually be identical to the output file that is the basis for the hand-coding,

[10] This is preferable to simply editing the CS output file itself because if something goes wrong, you still have the original. The `_h` is put at the end of the filename – crucially after the `.cod` or `.out` extension – so that one can more easily distinguish the two types of files visually in file listings and more simply manipulate them separately with wildcards. See the `mvcodh` script for a way to address the problem that this convention causes for subsequent searches with CS.

so an additional backup is redundant, and so you'll probably want to just delete it. If the user has chosen to move on to the next file, then `next` basically just starts over, looking for the next file without a matching `_h`.

Since it runs `codefinder`, `next` also looks for a file containing legal codes. It uses the same strategy as `codefinder`, first looking to see if the user has specified a file with the `--codesfile` option, then checking for a file named `codes` in the current directory. There is an option, `--last`, which tells `next` to work in reverse order, i.e. to start with the files that come last in ASCIIbetical order, and move towards those that come first. This is useful when two researchers are working on the same set of files, with one working front to back and the other back to front. Finally, there are two options that control the deletion of backup files. If you know that you always want to delete them, use `--yes`, and `next` will delete them without asking. If you know that you never want to delete them, `--no` tells `next` to leave them alone and not ask you.

One final thing to note about `next` is that the sole basis for its determination of which files are yet to be hand-coded is the existence of matching `_h` files. If you only get half way through a file and then quit for the day, when you run `next` to start your next session, it won't reopen that file, because as far as it knows, you're done with it. Running `codefinder` can come in handy in such cases. A strategy that has been used on our project is to delete the backup only when one is finished hand-coding the file. One can then see upon returning that a given file is still in progress.

2.5 progress v0.2.4

```
        Report on progress in hand-coding files in a given directory

Usage:
        progress [options] dir

Command-line options:
-c, --codesfile  <file>   Use <file> as list of known codes.

-b, --brief               Give brief report on a file-by-file basis,
                            not on a sentence-by-sentence basis
-s, --strict              Only count Known codes as done, not all
                            non-Machine codes
-v, --version             Print version information
-h, --help                Print this helpful message
```

The `progress` script gives a quick report of your progress in hand-coding a set of files. It depends on the coding conventions laid out for `codefinder` – in particular the use of `%` to indicate a Machine code that hasn't been hand checked yet – and on the file-naming conventions laid out for `next`. When run without any arguments or options, it reports on two measures of progress. First, it simply counts how many `_h` files there are, and how many `.out` or `.cod` files there are that don't have such a corresponding file. It lists the files in the latter category, reports the numbers for the two categories and gives a percentage. Then, it goes through all of the `.out` or `.cod` files and counts the number of invalid codes (defined below), then goes through the corresponding `_h` file if there is one and counts invalid codes, comparing the two totals to determine how many have been changed to valid codes.[11] It then reports these totals along with percentages. The end of the output for a

[11] This algorithm of comparing file-by-file is new in v0.2.4, and is meant to ensure proper handling of situations where hand-coding is done on files which are already partially hand-coded to start with – particularly files output by `integratecodes`.

large directory where more than half of the work has been done looks like this:

```
[...]
./coprefcath1.o3_1_2_oth_3.cod has no matching _h file
./coprefcath2.o3_1_2_oth_3.cod has no matching _h file
./coprefcura.o2_1_2_oth_3.cod has no matching _h file
./coprefgen.o3_1_2_oth_3.cod has no matching _h file

Total .cod files:                          100
Total .cod_h files:                         85 (85.0%)
To be done:                                 15 (15.0%)

Total sentences to hand-code:            16565
Sentences hand-coded so far:             11001 (66.4%)
To be done:                               5564 (33.6%)
```

By default, invalid codes means Machine codes. In other words, everything without a %, including Question codes and errors, is counted as having been done, i.e. as progress. If you want an estimate of your progress that's less lenient, use the `--strict` option. This causes only Known codes to be counted as valid. This of course requires the use of a file listing the legal codes, which is supplied in the same way as with `codefinder` and `next`. With the `--brief` option you can tell `progress` to skip the second part of the report based on actual clause counts, which can take a second or two when dealing with a full corpus and may be irrelevant if you're just trying to figure out what files you still have to do.

2.6 mvcodh v0.1.2

```
        For all files in current directory, change *.cod_h to *_h.cod
        and *.out_h to *_h.out.

Usage:
        mvcodh

Command-line options:
-h, --help                Print this helpful message
-v, --version             Print version information
```

Putting the `_h` at the very end of the filename of hand-coded files is very convenient while one is working on them and has a directory containing both CS output files and the corresponding hand-coded files, which need to be distinguished from one another. But of course it leads to a problem if you want to run further CS searches on the hand-coded files, because CS will only work with files ending in `.psd`, `.out` or `.cod`. The `mvcodh` script was thus created to move the `_h` suffix in front of the extensions after you're finished with the hand-coding and you've moved the hand-coded files into their own directory. It's the simplest and shortest script in the package, and has no options beyond the standard help and version. Note that it does not make new copies of the files with different names, it directly renames them. When it works properly, it produces no output.

2.7 integratecodes v0.2.3

```
        Change codes in input files on the basis of the codes in a changefile.
```

```
Usage:
        integratecodes [options] inputfiles

Command-line options:
-c, --changefile <file>   Use <file> as basis for changes
-D, --detailed            Print detailed info on the changes made
-d, --dir        <dir>    Use <dir> for output files
-h, --help                Print this helpful message
-l, --log        <file>   Record change information in <file>
-m, --missed              Report clauses in changefile which could not be
                            found in inputfiles
-n, --number              Print how many times each coding string was used
-p, --progress            Print progress indicator in terminal
                            in making changes
-r, --runcontrol <file>   Read <file> for configuation info
-s, --silent              Don't print anything
-t, --text                Allow change when id of sentence in changefile and
                            inputfile don't match perfectly, as long as their
                            texts do
-V, --verbose             Print detailed info on everything
-v, --version             Print version information
```

Change in v0.2.3: In order to be consistent with the other scripts, `-v` is now short for `--version`, while `-V` is the abbreviation for `--verbose`.

`integratecodes` is an extremely powerful script. It can be highly useful in certain situations, but also must be used with care. It takes one corpus file containing coding strings, called the changefile, and uses it as the basis for making changes in the coding strings in a series of other files, called the input files. Specifically, for every clause that appears in both the changefile and an input file, it changes the coding string in the input file to be the same as that in the changefile. A proper explanation of how this works and why it might be useful requires an example.

While coding perfect sentences according to the transitivity of the main verb, it became clear that we would have to add a new code. We had things like `trans` for clear transitives, `predo` for clauses with a predicate noun tagged as an object, `cogn` for clauses with a cognate object and `intrans` for everything that didn't fit into one of the categories. Clauses with a clausal object were not being treated specially. If the clausal object was an ECM clause, then we used the `ecm` code, but otherwise they were coded as the default, `intrans`. Since clauses of this sort, however, are clearly transitive in the sense relevant to us – i.e. the perfect auxiliary is never BE – we realized that this was not what we wanted, and introduced the code `clobj`. The problem was, we had already hand-coded nearly the entire ME corpus, weeks worth of work. Obviously we didn't want to re-read the entire thing. Now, it's not possible to reliably identify clausal objects structurally in a corpus file, because they aren't consistently annotated differently from adverbial embedded clauses, i.e. we can't just have CS find them and code them. But we can have CS find clauses with an embedded clause. These are unambiguously annotated, and after having CS search for them, we can hand-code them according to whether the clause in question is an object or not. This vastly reduced the number of sentences we had to re-read – something like two days' extra work instead of two or three weeks. The problem of course is, once this is done, you have a file which is correctly hand-coded, but covers only clauses containing embedded clauses, and then you have another set of files, the original hand-coded files, which contain all the clauses, but the ones with embedded clauses are not correctly coded.

This is what `integratecodes` was written for. It took the file containing the clausal objects as the changefile and the whole hand-coded corpus as the input files, and changed the codes in the latter to be in line with the former. I.e. it updated the coding in the full corpus to reflect the changes that were made in the embedded clause subset.

It's important to understand how `integratecodes` identifies clauses in order to change their coding strings. Each sentence in a corpus file will have an id number, e.g. something like `CMAELR3,30.123`, and `integratecodes` starts by checking to see whether the id of a clause in the changefile is matched by any of the clauses in the input files. However, an id match is not sufficient to ensure that we're actually dealing with the same clause, because the numbers are given to sentences, not to clauses. I.e. every clause in a multi-clausal sentence will have the same id number. To ensure that it changes the code in the correct clause of a given sentence, `integratecodes` thus also compares the text of the clauses. Specifically, it takes the tree structure of a clause, extracts all of the lowercase letters, and puts them together in a string which can be used for comparison.[12] If both the id number and the text of a clause in an input file match with a clause in the changefile, the coding string on the clause in the input file is changed to match that in the changefile.

Input files must be specified as command-line arguments. The changefile can either be supplied with the `--changefile` option, or interactively. `integratecodes` requires that you specify a directory for it to put its output files in. This is for the simple reason that the output files have the same names as the input files, so trying to put them in the same directory would be bad. The output directory can be given with the `--dir` option, or will be queried for interactively.

There is a series of options that control what `integratecodes` reports about its actions. By default, it prints basic numbers on its success: how many clauses are in the changefile, and how many changes were made. If the clauses in the changefile are a subset of those in the input files, then these two numbers should be the same, i.e. it should change every clause that occurs in the changefile.[13] If no changes were made successfully, it reports that instead. If even this spare output is too much for you, you can use the `--silent` option, and `integratecodes` won't say a thing. Usually, though, you actually want more detail on what happened, to make sure that things went OK. There are three additional kinds of info that `integratecodes` can report. With the `--detailed` option,[14] you tell the script to print out information on each individual change made, including the id number of the clause, the coding string it had in the input file, and the coding string it has in the changefile and the output file. The `--number` flag triggers a report on which coding strings were used in making changes and how often. Finally, the `--missed` flag tells `integratecodes` to report on clauses in the changefile that could not successfully be found in the input files, resulting in a missed change. Under normal circumstances, there shouldn't be any misses, but it's always good to check for them. As we'll see below, there are some special circumstances where this becomes very important. If you want all three kinds of information, use the `--verbose` flag, which turns the three just discussed on. All four of these flags override `--silent`.

If you run `integratecodes` on a large number of sentences, the reports generated will be quite large, and you'll want to save them. You can send them off to a file instead of the terminal with `--log` followed by a file name. This is better than just using the shell to redirect the output because it doesn't break the interactive parts of the script, and it still will print some basic info to the screen so that you know whether it worked or not. Now, because of the complexity of what it has to do, `integratecodes` takes quite a while to run. Since the reports can't be generated until the end, this means you may be looking at a blank screen for quite some time before you know if anything's even

[12] The tree structure is used instead of the urtext because the urtext again often contains the entire sentence, not just the relevant clause. Extracting only the lowercase letters is just a simple way to get the text out from the parentheses and labels.

[13] Even if the coding string for a clause was the same to begin with in the changefile and the input file, if the two are successfully matched, `integratecodes` counts it as a change. It always overwrites the coding string, even if vacuously.

[14] Note that its short form is `-D`, since `-d` is already taken as the short form of `--dir`

happening. If this bothers you, use the `--progress` flag, and the program will keep you abreast of what it's doing with a ridiculously simple progress meter.[15] This doesn't seem to slow things down appreciably, so there's no harm in using it.

Beyond the scenario discussed above, `integratecodes` has a number of less obvious potential uses. One is to propagate coding changes backwards through a series of corpus files. I.e. if you have a series of files that are the output of a series CS searches and coding runs on the same file(s), and you find that something has been miscoded, or change your mind about some coding question, you can make the changes in one file in the series (say `cmwycer.m3_1_2_h.cod`) and use integratecodes to have the changes matched in every other file in the series (say `cmwycser.m3_1_2_h_3.out` and `cmwycser.m3_1_2_h_3_4.out`).

The most ambitious and obscure use to which we've put the program was in dealing with different versions of the PPCEME. We had run a series of searches on, and then hand-coded, a pre-release version of the corpus. Subsequent to that, a number of changes were made in getting the corpus ready for release. Some were minor, but there were several corrections of errors that were directly relevant to the stuff we were interested in, so it became clear that we would have to rerun the searches on the final release version when it came out. But of course this implied completely redoing the hand-coding, again weeks worth of work. We realized, though, that if we played around with things a bit, we could transfer the hand-coding we had done on the older version of the corpus onto the outputs from the new version with `integratecodes`.

There was one snag, though: one of the changes in the corpus involved the way that id numbers were assigned, with the result that nearly every sentence had a different id in the new version than it had had in the old. I.e. as it stood `integratecodes` would fail to find matches for most of the sentences and thus fail to make the changes. Fortunately, they had not changed completely. A sentence id is made up of three parts: first the name of the text/file the sentence is from (potentially including period information), then the page number and potentially a chapter or section number, and finally the sequential number of the sentence within the file. So the id `ZOUCH-E3-P2,158.4` breaks down as follows:

File:	ZOUCH-E3-P2
Page:	158
Sent. No.:	4

The changes in the corpus had only affected the latter in most cases, so we could still use the first two parts, combined with the text of the actual clause, and be pretty certain that we wouldn't get any false matches.

So we added an additional option to `integratecodes`, `--text`, which tells it to go ahead and make a change as long as the text of two clauses being compared matches, and the ids match partially in the way just described. Now, this introduces some inaccuracy into the proceedings, and a number of matches will be missed. This is where the detailed reports about what changes were made and what changes were missed come in very handy. Also, when you use the `--text` option, the report on the change made triggered by the `--detailed` flag will distinguish between changes made on the basis of a perfect id match, and those made on the basis of an imperfect one. `integratecodes` with the `--text` option is extremely powerful and should be used with care. It is of course entirely unnecessary under normal circumstances, where the corpus you're dealing with hasn't changed. But when it's required, it can be very helpful. In our case, well over 80% of the changes were made successfully, saving us from redoing a couple weeks' worth of work.

An additional mechanism for handling such instances has been added in v0.2.3: substitution rules applied to id strings. In the development of the PPCEME corpus discussed above, a few changes

[15] All it really tells you is that `integratecodes` is making progress, not how long it will be until it's finished. Still, this is enough to indicate that the program hasn't frozen up.

were made in the determination of id strings for certain texts. E.g., what had originally been labelled as `(HOOKERSERM1...)` came to be labelled as `(ID HOOKER-A...)`. This obviously causes problems for `integratecodes`. To deal with this, what you need is a way to specify substitutions that the program can do before comparing id strings so as not to throw out what are real matches. In order to do this, you can now supply a file containing substitution commands with the `--runcontrol` flag. Here's an example of such a file, demonstrating the syntax:

```
# following ditches -E3-P1 style infixes in new versions of EME corpus files
input: s/(.*)-E\d+-\w\d?(,.*)/$1$2/

# the following deal with change in names of hooker sermon ids:
change: s/HOOKERSERM1/HOOKER-A/
change: s/HOOKERSERM2/HOOKER-B/

# the following deal with change in names of spencer texts:
input: s/SPENCER-\d{4}/SPENCER/
```

Lines starting with # are treated as comments and ignored.[16] Commands begin with a word indicating their scope: `input:` for substitutions to carry out on id strings in the input files, `change:` for substitutions to carry out on id strings in the changefile, and `all:` for substitutions to carry out on id strings in all files (not instantiated in the above example). Whatever follows the scope indicator will be used as a perl substitution command. I.e. it should be of the form `s/`*REGEX*`/`*STRING*`/`, where *REGEX* is a regular expression describing strings to be replaced and *STRING* is the string to replace them with, potentially containing perl match variables like `$1`, `$2`.

The runcontrol file above contains three substitution commands. The first gets rid of strings like `-E3-P1` or `-E2-H` that have been added to id strings in the new version of the PPCEME. These indicate the period of the text and the portion of the corpus from which it is taken, and were not present in the id tags of early versions of the corpus. The second handles the change described above, treating ids in the changefile containing `HOOKERSERM1` as though they contained `HOOKER-A`, and those with `HOOKERSERM2` as `HOOKER-B`. The last ignores the years (i.e. four digits) that have been added to id tags in the Spencer texts. Note, crucially, that these substitutions are **not** carried out on the text in any of the actual files, i.e. the id strings in the output files are completely unaffected. They are applied only to the representation of the id strings internal to `integratecodes` that the program is using to determine whether a sentence in an input file matches a sentence in the changefile.

2.8 ipcoding v0.1.2

```
        Change old style coding files to new style ones, i.e. move CODING
        node inside IP.

Usage:
        ipcoding [options] inputfiles

Command-line options:
-d, --dir        <dir>    Use <dir> for output files
-h, --help                Print this helpful message
-v, --version             Print version information
```

[16] Inline comments following # in a non-line-initial position are not currently supported.

The final script that deals with the hand-coding process, and one of the simplest, is `ipcoding`. One of the changes between versions 1 and 2 of CS is in how the coding string is placed in the tree structure of a clause. In version 1, the `CODING` node was place outside the `IP` node:

```
(0 NODE (0 CODING intrans)
        (1 IP-SUB (2 NP-SBJ (3 PRO +tey))
                  (4 HVD hadde)
                  (5 DON don)
                  (6 ADVP (7 ADV +tus)))
        (8 ID CMWYCSER,240.314))
```

In version 2, on the other hand, it is placed **inside** `IP`:

```
(NODE (IP-SUB (CODING intrans)
              (NP-SBJ (PRO +tey))
              (HVD hadde)
              (DON don)
              (ADVP (ADV +tus)))
      (ID CMWYCSER,240.314))
```

This presents a bit of a problem if you have output files left over from version 1 which you want to run additional searches on with version 2. The latter will not accept the former as input, and if what you're dealing with are hand-coded files, you can't just re-create the files by redoing all the searches with version 2. This is perhaps another instance where `integratecodes` could be used, but `ipcoding` is a much simpler solution. It takes a series of input files, named as command-line arguments, which have the old coding format, and outputs a series of files which are identical, except for having the coding strings moved inside of `IP`. Like `integratecodes`, it creates new copies of the files with the same names, so it requires an output directory to put them in, again either specified with the `--dir` flag or queried for interactively.

2.9 tagfinder v0.2.2

```
        Find all forms in a corpus file with POS tags matching
        a regular expression.

Usage:
        tagfinder [options] file(s)

Command-line options:
-h, --help                Print this helpful message.
-l, --linenum             Print the linenumbers on which each form appears.
-n, --number              Sort forms by number of times they appear.
-t, --tag        <regex>  Search POS tags matching <regex>
-v, --version             Print version information.
```

The `tagfinder` script searches a specified corpus file or files for part-of-speech tags which contain a string matching a specified regular expression, and reports on what forms appear within those nodes. That is, if you want to know what perfect participle forms appear in a given set of files, you can run `tagfinder` to report on forms tagged with `VBN`. This is very similar to the lexicon function in version 2 of CS, but the two differ substantially in the details. The lexicon function is far more flexible, allowing you to create a lexicon for all words in a file, or for all words of a specific form,

rather than just all words with a particular POS tag. On the other hand, `tagfinder` allows you to use regular expressions in specifying the POS label (though to be fair it is somewhat difficult to imagine a scenario where the wildcards provided by CS won't be sufficient for this task and you'll actually need the added power of regular expressions) and is capable of reporting the line numbers on which particular forms are found.

The default output is as follows, with the forms listed in alphabetical order, followed by the number of times they appeared in the text(s);

```
-------------------------------------
Output of tagfinder: Mon Mar 28 15:41:18 2005

axid (1)
come (3)
comun (6)
entrid (1)
etun (1)
gon (2)
herd (1)
herde (1)
lerned (2)
.
.
.
-------------------------------------
Total forms: 29
```

The `--number` option causes the forms to be sorted by their frequency, rather than alphabetically:

```
-------------------------------------
Output of tagfinder: Mon Mar 28 15:44:18 2005

comun (6)
come (3)
seid (2)
seyn (2)
lerned (2)
gon (2)
risun (1)
takun (1)
stonde (1)
.
.
.
-------------------------------------
Total forms: 29
```

With the `--linenum` flag, you can make `tagfinder` report the line numbers on which the forms appear, rather than just the number of times:

```
axid: 157
```

```
come: 302 456 529

comun: 111 175 252 285 548 565

entrid: 214

etun: 270

gon: 130 376

herd: 234

herde: 409
.
.
.
```

Note that the output with `--linenum` is always in alphabetical order. The `--number` flag is ignored. As with `codefinder`, you'll have to add anchors to your regex if you want exact matches. E.g. if you want only `VBN`, not things like `VBN21`, use `tagfinder --tag 'VBN$'`.

2.10 editcode v0.1.0

```
        Search for codestrings matching a specified regular expression
        in specified corpus files, and edit them in turn.

Usage:
        editcode regex [file(s)]

Command-line options:
-f, --force              Don't prompt before continuing to next example
-h, --help               Print this helpful message
-v, --version            Print version information
```

`editcode` is a new addition in CsPTools v0.3.1. It is a very simple program that automates the process of examining lots of sentences with the same or similar coding strings. This is useful e.g. if you decide you want to change how you use a particular coding string, or if you want to read all the sentences with a particular type of code to manually look for a pattern. The program takes as its first argument (obligatorily) a regular expression describing coding strings, and the files to look in as optional additional arguments. It then searches through those files, finds all sentences with codes matching the regular expression, and opens the files to the appropriate positions for the examples, one after another in emacs. If no files are specified, the default is to examine all files in the current directory. By default, after you examine or edit a particular example and close emacs, `editcode` will print a prompt, giving the option to enter 'q' to quit, or ENTER to continue to the next example. This behavior can be overridden – forcing `editcode` to immediately open the next example without prompting – with the `--force` flag. The reason for the default prompting is that it would otherwise be rather difficult to quit `editcode` before it finished going through all the specified files, say if you had made an error in specifying the regex.[17]

[17]Since all `editcode` really does is fire off a series of emacs invocations, it is only ever in the foreground for fractions of a second. I.e. if you wanted to quit, and hit ctrl-c, the interrupt would almost always be intercepted by the currently

open emacs. If you quit the current emacs, the next one will open before you get a chance to hit ctrl-c, so basically you'd have to either let `editcode` run its course – potentially very long if your regex matches a lot of hits – or you'd have to open another terminal and kill `editcodes` from there.