

Hunspell – The free spelling checker

About Hunspell

Hunspell is a spell checker and morphological analyzer library and program designed for languages with rich morphology and complex word compounding or character encoding. Hunspell interfaces: Ispell-like terminal interface using Curses library, Ispell pipe interface, OpenOffice.org UNO module.

Main features of Hunspell spell checker and morphological analyzer:

- Unicode support (affix rules work only with the first 65535 Unicode characters)
- Morphological analysis (in custom item and arrangement style) and stemming
- Max. 65535 affix classes and twofold affix stripping (for agglutinative languages, like Azeri, Basque, Estonian, Finnish, Hungarian, Turkish, etc.)
- Support complex compoundings (for example, Hungarian and German)
- Support language specific features (for example, special casing of Azeri and Turkish dotted i, or German sharp s)
- Handle conditional affixes, circumfixes, fogemorphemes, forbidden words, pseudoroots and homonyms.
- Free software (LGPL, GPL, MPL tri-license)

Usage

The src/tools dictionary contains ten executables after compiling (or some of them are in the src/win_api):

affixcompress: dictionary generation from large (millions of words) vocabularies
analyze: example of spell checking, stemming and morphological analysis
chmorph: example of automatic morphological generation and conversion
example: example of spell checking and suggestion
hunspell: main program for spell checking and others (see manual)
hunzip: decompressor of hzip format
hzip: compressor of hzip format
makealias: alias compression (Hunspell only, not back compatible with MySpell)
munch: dictionary generation from vocabularies (it needs an affix file, too).
unmunch: list all recognized words of a MySpell dictionary

After compiling and installing (see INSTALL) you can run the Hunspell spell checker (compiled with user interface) with a Hunspell or MySpell dictionary:

```
hunspell -d en_US text.txt
```

or without interface:

```
hunspell  
hunspell -d en_UK -l <text.txt
```

Dictionaries consist of an affix and dictionary file, see tests/ or <http://wiki.services.openoffice.org/wiki/Dictionaries>.

Using Hunspell library with GCC

Including in your program:

```
#include <hunspell/hunspell.h>
```

Linking with Hunspell static library:

```
g++ -lhunspell example.cxx
```

Dictionaries

MySpell & Hunspell dictionaries: <http://wiki.services.openoffice.org/wiki/Dictionaries>

Aspell dictionaries (need some conversion): <ftp://ftp.gnu.org/gnu/aspell/dict>

Conversion steps: see relevant feature request at <http://hunspell.sf.net>.

László Németh
nemeth at Oo

Table of Contents

Hunspell usage notes.....	3
NAME.....	3
SYNOPSIS.....	3
DESCRIPTION.....	3
OPTIONS.....	4
EXAMPLES.....	8
ENVIRONMENT.....	8
FILES.....	9
SEE ALSO.....	9
AUTHOR.....	9
BUGS.....	9
Hunspell dictionary development.....	10
NAME.....	10
DESCRIPTION.....	10
GENERAL OPTIONS.....	11
OPTIONS FOR SUGGESTION.....	12
OPTIONS FOR COMPOUNDING.....	15
OPTIONS FOR AFFIX CREATION.....	17
OTHER OPTIONS.....	18
Morphological analysis.....	20
Optional data fields.....	20
Twofold suffix stripping.....	22
Extended affix classes.....	23
Homonyms.....	23
Prefix--suffix dependencies.....	24
Circumfix.....	25
Compounds.....	26
Unicode character encoding.....	28
SEE ALSO.....	29
Hunspell API.....	30
NAME.....	30
SYNOPSIS.....	30
DESCRIPTION.....	31
EXAMPLE.....	33
AUTHORS.....	33

Hunspell usage notes



NAME

hunspell – spell checker, stemmer and morphological analyzer



SYNOPSIS

```
hunspell [-1aDGHhLlmnstvw] [--check-url] [-d dict[,dict2,...]] [--help] [-i enc] [-p dict] [-vv] [--version] [file(s)]
```



DESCRIPTION

Hunspell is fashioned after the *Ispell* program. The most common usage is "hunspell" or "hunspell filename". Without filename parameter, hunspell checks the standard input. Typing "cat" and "example" in two input lines, we got an asterisk (it means "cat" is a correct word) and a line with corrections:



```
$ hunspell -d en_US
Hunspell 1.2.3
*
& example 4 0: example, examples, ex sample, ex-sample
```



Correct words signed with an '*', '+' or '-', unrecognized words signed with '#' or '&' in output lines (see later). (Close the standard input with Ctrl-d on Unix/Linux and Ctrl-Z Enter or Ctrl-C on Windows.) With filename parameters, *hunspell* will display each word of the files which does not appear in the dictionary at the top of the screen and allow you to change it. If there are "near misses" in the dictionary, then they are also displayed on following lines. Finally, the line containing the word and the previous line are printed at the bottom of the screen. If your terminal can display in reverse video, the word itself is highlighted. You have the option of replacing the word completely, or choosing one of the suggested words. Commands are single characters as follows (case is ignored):



- R Replace the misspelled word completely.
- Space Accept the word this time only.
- A Accept the word for the rest of this *hunspell* session.
- I Accept the word, capitalized as it is in the file, and update private dictionary.
- U Accept the word, and add an uncapitalized (actually, all lower-case) version to the private dictionary.

- S Ask a stem and a model word and store them in the private dictionary. The stem will be accepted also with the affixes of the model word.
- 0-n Replace with one of the suggested words.
- X Write the rest of this file, ignoring misspellings, and start next file.
- Q Exit immediately and leave the file unchanged.
- ^Z Suspend hunspell.
- ? Give help screen.



OPTIONS

- Check only first field in lines (delimiter = tabulator).
- 1

- The **-a** option is intended to be used from other programs through a pipe. In this mode, *hunspell* prints a one-line version identification message, and then begins reading lines of input. For each input line, a single line is written to the standard output for each word checked for spelling on the line. If the word was found in the main dictionary, or your personal dictionary, then the line contains only a '*''. If the word was found through affix removal, then the line contains a '+', a space, and the root word. If the word was found through compound formation (concatenation of two words, then the line contains only a '-'.
 - a**

If the word is not in the dictionary, but there are near misses, then the line contains an '&', a space, the misspelled word, a space, the number of near misses, the number of characters between the beginning of the line and the beginning of the misspelled word, a colon, another space, and a list of the near misses separated by commas and spaces.

Also, each near miss or guess is capitalized the same as the input word unless such capitalization is illegal; in the latter case each near miss is capitalized correctly according to the dictionary.

Finally, if the word does not appear in the dictionary, and there are no near misses, then the line contains a '#', a space, the misspelled word, a space, and the character offset from the beginning of the line. Each sentence of text input is terminated with an additional blank line, indicating that *hunspell* has completed processing the input line.

These output lines can be summarized as follows:



OK: *



Root: + <root>



Compound:



–

Miss: & <original> <count> <offset>: <miss>, <miss>, ...



None: # <original> <offset>



For example, a dummy dictionary containing the words "fray", "Frey", "fry", and "refried" might produce the following response to the command "echo `frqy refries | hunspell -a`":



```
(#) Hunspell 0.4.1 (beta), 2005-05-26
& frqy 3 0: fray, Frey, fry
& refries 1 5: refried
```

This mode is also suitable for interactive use when you want to figure out the spelling of a single word (but this is the default behavior of *hunspell* without `-a`, too).

When in the `-a` mode, *hunspell* will also accept lines of single words prefixed with any of `'*`, `'&`, `'@`, `'+`, `'-`, `'~`, `'#`, `'!`, `'%`, `'"`, or `'^`. A line starting with `'*` tells *hunspell* to insert the word into the user's dictionary (similar to the `I` command). A line starting with `'&` tells *hunspell* to insert an all-lowercase version of the word into the user's dictionary (similar to the `U` command). A line starting with `'@` causes *hunspell* to accept this word in the future (similar to the `A` command). A line starting with `'+`, followed immediately by `tex` or `nroff` will cause *hunspell* to parse future input according to the syntax of that formatter. A line consisting solely of a `'+` will place *hunspell* in TeX/LaTeX mode (similar to the `-t` option) and `'-` returns *hunspell* to `nroff/troff` mode (but these commands are obsolete). However, the string character type is *not* changed; the `'~` command must be used to do this. A line starting with `'~` causes *hunspell* to set internal parameters (in particular, the default string character type) based on the filename given in the rest of the line. (A file suffix is sufficient, but the period must be included. Instead of a file name or suffix, a unique name, as listed in the language affix file, may be specified.) However, the formatter parsing is *not* changed; the `'+` command must be used to change the formatter. A line prefixed with `'#` will cause the personal dictionary to be saved. A line prefixed with `'!` will turn on *terse* mode (see below), and a line prefixed with `'%` will return *hunspell* to normal (non-*terse*) mode. A line prefixed with `'"` will turn on verbose-correction mode (see below); this mode can only be disabled by turning on *terse* mode with `'%`.

Any input following the prefix characters '+' , '-' , '#', '!', '%', or '^' is ignored, as is any input following the filename on a '~' line. To allow spell-checking of lines beginning with these characters, a line starting with '^' has that character removed before it is passed to the spell-checking code. It is recommended that programmatic interfaces prefix every data line with an uparrow to protect themselves against future changes in *hunspell*.

To summarize these:

* Add to personal dictionary



- @ Accept word, but leave out of dictionary
- # Save current personal dictionary
- ~ Set parameters based on filename
- + Enter TeX mode
- Exit TeX mode
- ! Enter terse mode
- % Exit terse mode
- ^ Enter verbose-correction mode
- ^ Spell-check rest of line

In *terse* mode, *hunspell* will not print lines beginning with



'*', '+', or '-', all of which indicate correct words. This significantly improves running speed when the driving program is going to ignore correct words anyway.



In *verbose-correction* mode, *hunspell* includes the original word immediately after the indicator character in output lines beginning with '*', '+', and '-', which simplifies interaction for some programs.



—**check-url**

Check URLs, e-mail addresses and directory paths.



- Show detected path of the loaded dictionary, and list of the search path and the available dictionaries.

D



-d dict,dict2,...



Set dictionaries by their base names with or without paths. Example of the syntax:



`-d en_US,en_geo,en_med,de_DE,de_med`



`en_US` and `de_DE` are base dictionaries, they consist of `aff` and `dic` file pairs: `en_US.aff`, `en_US.dic` and `de_DE.aff`, `de_DE.dic`. `en_geo`, `en_med`, `de_med` are special dictionaries: dictionaries without affix file. Special dictionaries are optional extension of the base dictionaries usually with special (medical, law etc.) terms. There is no naming convention for special dictionaries, only the `".dic"` extension: dictionaries without affix file will be an extension of the preceding base dictionary (right order of the parameter list needs for good suggestions). First item of `-d` parameter list must be a base dictionary.



-G Print only correct words or lines.

G

-H The input file is in SGML/HTML format.

H



-h, --help



Short help.



-i enc Set input encoding.

-L Print lines with misspelled words.

-l The "list" option is used to produce a list of misspelled words from the standard input.

-m Analyze the words of the input text (see also `hunspell(4)` about morphological analysis). Without dictionary morphological data, signs the flags of the affixes of the word forms for dictionary developers.

-n The input file is in `nroff/troff` format.



-P password



Set password for encrypted dictionaries.



-p dict



Set path of personal dictionary. Default dictionary depends from the locale settings. Without locale support, the default personal dictionary is the `$HOME/.hunspell_default`.



Using `-d` or the `DICTIONARY` environmental variable, personal dictionary will be **`$HOME/.hunspell_dicname`**



-s Stem the words of the input text (see also `hunspell(4)` about stemming). It depends from the dictionary data.

-t The input file is in TeX or LaTeX format.



-v, --version



Print version number.



-vv Print `ispell(1)` compatible version number.

-w Print misspelled words (= lines) from one word/line input.

EXAMPLES



`hunspell -d en_US english.html`



`hunspell -d en_US,en_US_med medical.txt`



`hunspell -d ~/openoffice.org2.4/share/dict/ooo/de_DE`



`hunspell *.html`



`hunspell -l text.html`

ENVIRONMENT



DICTIONARY

Similar to `-d`.



DICPATH

Dictionary path.



WORDLIST

Equivalent to `-p`.





FILES

`/usr/share/myspell/default.aff` Path of default affix file. See `hunspell(4)`.



`/usr/share/myspell/default.dic` Path of default dictionary file. See `hunspell(4)`.



`~/.hunspell_default`. Default path to personal dictionary.



SEE ALSO

`hunspell(3)`, `hunspell(4)`



AUTHOR

Author of Hunspell executable is László Németh. For Hunspell library, see `hunspell(3)`.



`ispell` manual based on `Ispell`'s manual. See `ispell(1)`.



BUGS

There are some layout problems with long lines.

Hunspell dictionary development



NAME

hunspell – format of Hunspell dictionaries and affix files



DESCRIPTION

Hunspell(1) requires two files to define the language that it is spell checking. The first file is a dictionary containing words for the language, and the second is an "affix" file that defines the meaning of special flags in the dictionary.



A dictionary file (*.dic) contains a list of words, one per line. The first line of the dictionaries (except personal dictionaries) contains the approximate word count (for optimal hash memory size). Each word may optionally be followed by a slash ("/") and one or more flags, which represents affixes or special attributes. Dictionary words can contain also slashes with the "" syntax. Default flag format is a single (usually alphabetic) character. In a Hunspell dictionary file, there are also optional fields separated by tabulators or spaces (spaces from Hunspell 1.2), see Optional data fields.



Personal dictionaries are simple word lists. Asterisk at the first character position signs prohibition. A second word separated by a slash sets the affixation.



```
foo
Foo/Simpson
*bar
```



In this example, "foo" and "Foo" are personal words, plus Foo will be recognized with affixes of Simpson (Foo's etc.) and bar is a forbidden word.



An affix file (*.aff) may contain a lot of optional attributes. For example, **SET** is used for setting the character encodings of affixes and dictionary files. **TRY** sets the change characters for suggestions. **REP** sets a replacement table for multiple character corrections in suggestion mode. **PFX** and **SFX** defines prefix and suffix classes named with affix flags.



The following affix file example defines UTF-8 character encoding. 'TRY' suggestions differ from the bad word with an English letter or an apostrophe. With these REP definitions, Hunspell can suggest the right word form, when the misspelled word contains f instead of ph and vice versa.



```
SET UTF-8
TRY esianrtolcdugmphbyfvkwzESIANRTOLCDUGMPHBYFVKWZ'

REP 2
REP f ph
REP ph f
```

```
PFX A Y 1
PFX A 0 re .

SFX B Y 2
SFX B 0 ed [^y]
SFX B y ied y
```

There are two affix classes in the dictionary. Class A defines a ‘re-’ prefix. Class B defines two ‘-ed’ suffixes. First suffix can be added to a word if the last character of the word isn’t ‘y’. Second suffix can be added to the words terminated with an ‘y’. (See later.) The following dictionary file uses these affix classes.

```
3
hello
try/B
work/AB
```

All accepted words with this dictionary: "hello", "try", "tried", "work", "worked", "rework", "reworked".

GENERAL OPTIONS

Hunspell source distribution contains more than 80 examples for option usage.

SET encoding

Set character encoding of words and morphemes in affix and dictionary files. Possible values: UTF-8, ISO8859-1 – ISO8859-10, ISO8859-13 – ISO8859-15, KOI8-R, KOI8-U, microsoft-cp1251, ISCII-DEVANAGARI.

FLAG value

Set flag type. Default type is the extended ASCII (8-bit) character. ‘UTF-8’ parameter sets UTF-8 encoded Unicode character flags. The ‘long’ value sets the double extended ASCII character flag type, the ‘num’ sets the decimal number flag type. Decimal flags numbered from 1 to 65000, and in flag fields are separated by comma. BUG: UTF-8 flag type doesn’t work on ARM platform.

COMPLEXPREFIXES

Set twofold prefix stripping (but single suffix stripping) for agglutinative languages with right-to-left writing system.

LANG langcode

Set language code. In Hunspell may be language specific codes enabled by LANG code. At present there are az_AZ, hu_HU, TR_tr specific codes in Hunspell (see the source code).

IGNORE characters



Ignore characters from dictionary words, affixes and input words. Useful for optional characters, as Arabic diacritical marks (Harakat).



AF number_of_flag_vector_aliases



AF flag_vector



Hunspell can substitute affix flag sets with ordinal numbers in affix rules (alias compression, see makealias tool). First example with alias compression:



```
3
hello
try/1
work/2
```



AF definitions in the affix file:



```
SET UTF-8
TRY esianrtolcdugmphbyfvkwzESIANRTOLCDUGMPHBYFVKWZ'
AF 2
AF A
AF AB
```



It is equivalent of the following dic file:



```
3
hello
try/A
work/AB
```



See also tests/alias* examples of the source distribution.



Note: If affix file contains the FLAG parameter, define it before the AF definitions.



Note II: Use makealias utility in Hunspell distribution to compress aff and dic files.



AM number_of_morphological_aliases



AM morphological_fields



Hunspell can substitute also morphological data with ordinal numbers in affix rules (alias compression). See tests/alias* examples.



OPTIONS FOR SUGGESTION

Suggestion parameters can optimize the default n-gram, character swap and deletion suggestions of Hunspell. REP is suggested to fix the typical and especially bad language specific bugs, because the REP suggestions have the highest priority in the suggestion list.

PHONE is for languages with not pronunciation based orthography.

KEY characters_separated_by_vertical_line_optionally

Hunspell searches and suggests words with one different character replaced by a neighbor KEY character. Not neighbor characters in KEY string separated by vertical line characters. Suggested KEY parameters for QWERTY and Dvorak keyboard layouts:

```
KEY qwertyuiop|asdfghjkl|zxcvbnm  
KEY pyfgcrll|aeouidhtns|qjkbmwwz
```

Using the first QWERTY layout, Hunspell suggests "nude" and "node" for "*nide". A character may have more neighbors, too:

```
KEY qwertzuop|yxcvbnm|qaw|say|wse|dsx|sy|edr|fdc|dx|rft|gfv|fc|tgz|hgb|gv|zhu|jhn|hb|uji|lkm
```

TRY characters

Hunspell can suggest right word forms, when they differ from the bad input word by one TRY character. The parameter of TRY is case sensitive.

NOSUGGEST flag

Words signed with NOSUGGEST flag are not suggested. Proposed flag for vulgar and obscene words (see also SUBSTANDARD).

MAXNGRAMSUGS num

Set number of n-gram suggestions. Value 0 switches off the n-gram suggestions.

NOSPLITSUGS

Disable split-word suggestions.

SUGSWITHDOTS

Add dot(s) to suggestions, if input word terminates in dot(s). (Not for OpenOffice.org dictionaries, because OpenOffice.org has an automatic dot expansion mechanism.)

REP number_of_replacement_definitions

REP what replacement

We can define language-dependent phonetic information in the affix file (.aff) by a replacement table. First REP is the header of this table and one or more REP data line are following it. With this table, Hunspell can suggest the right

forms for the typical faults of spelling when the incorrect form differs by more, than 1 letter from the right form. For example a possible English replacement table definition to handle misspelled consonants:

```
REP 8
REP f ph
REP ph f
REP f gh
REP gh f
REP j dg
REP dg j
REP k ch
REP ch k
```

Note I: It's very useful to define replacements for the most typical one-character mistakes, too: with REP you can add higher priority to a subset of the TRY suggestions (suggestion list begins with the REP suggestions).

Note II: Suggesting separated words by REP, you can specify a space with an underline:

```
REP 1
REP alot a_lot
```

Note III: Replacement table can be used for a stricter compound word checking (forbidding generated compound words, if they are also simple words with typical fault, see CHECKCOMPOUNDREP).

MAP number_of_map_definitions

MAP string_of_related_chars

We can define language-dependent information on characters that should be considered related (i.e. nearer than other chars not in the set) in the affix file (.aff) by a character map table. With this table, Hunspell can suggest the right forms for words, which incorrectly choose the wrong letter from a related set more than once in a word. For example a possible mapping could be for the German umlauted ü versus the regular u; the word Frühstück really should be written with umlauted u's and not regular ones

```
MAP 1
MAP uü
```

PHONE number_of_phone_definitions

PHONE what replacement

PHONE uses a table-driven phonetic transcription algorithm borrowed from Aspell. It is useful for languages with not pronunciation based orthography. You can add a full alphabet conversion and other rules for conversion of special letter sequences. For detailed documentation see <http://aspell.net/man-html/Phonetic-Code.html>. Note: Multibyte UTF-8 characters have not worked with bracket expression yet. Dash expression has signed bytes and not UTF-8

characters yet.

OPTIONS FOR COMPOUNDING

BREAK number_of_break_definitions

BREAK character_or_character_sequence

Define break points for breaking words and checking word parts separately. Rationale: useful for compounding with joining character or strings (for example, hyphen in English and German or hyphen and n-dash in Hungarian). Dashes are often bad break points for tokenization, because compounds with dashes may contain not valid parts, too.) With BREAK, Hunspell can check both side of these compounds, breaking the words at dashes and n-dashes:

```
BREAK 2
BREAK -
BREAK -- # n-dash
```

Breaking are recursive, so foo-bar, bar-foo and foo-foo--bar-bar would be valid compounds.

Note: COMPOUNDRULE is better (or will be better) for handling dashes and other compound joining characters or character strings. Use BREAK, if you want check words with dashes or other joining characters and there is no time or possibility to describe precise compound rules with COMPOUNDRULE (COMPOUNDRULE has handled only the last suffixation of the compound word yet).

Note II: For command line spell checking, set WORDCHARS parameters: WORDCHARS --- (see tests/break.*) example

COMPOUNDRULE number_of_compound_definitions

COMPOUNDRULE compound_pattern

Define custom compound patterns with a regex-like syntax. The first COMPOUNDRULE is a header with the number of the following COMPOUNDRULE definitions. Compound patterns consist compound flags and star or question mark meta characters. A flag followed by a '*' matches a word sequence of 0 or more matches of words signed with this compound flag. A flag followed by a '?' matches a word sequence of 0 or 1 matches of a word signed with this compound flag. See tests/compound*.* examples.

Note: '*' and '?' metacharacters work only with the default 8-bit character and the UTF-8 FLAG types.

Note II: COMPOUNDRULE flags haven't been compatible with the COMPOUNDFLAG, COMPOUNDBEGIN, etc. compound flags yet (use these flags on different words).



COMPOUNDMIN num

Minimum length of words in compound words. Default value is 3 letters.



COMPOUNDFLAG flag

Words signed with COMPOUNDFLAG may be in compound words (except when word shorter than COMPOUNDMIN). Affixes with COMPOUNDFLAG also permits compounding of affixed words.



COMPOUNDBEGIN flag

Words signed with COMPOUNDBEGIN (or with a signed affix) may be first elements in compound words.



COMPOUNDLAST flag

Words signed with COMPOUNDLAST (or with a signed affix) may be last elements in compound words.



COMPOUNDMIDDLE flag

Words signed with COMPOUNDMIDDLE (or with a signed affix) may be middle elements in compound words.



ONLYINCOMPOUND flag

Suffixes signed with ONLYINCOMPOUND flag may be only inside of compounds (Fuge-elements in German, fogemorphemes in Swedish). ONLYINCOMPOUND flag works also with words (see tests/onlyincompound.*).



COMPOUNDPERMITFLAG flag

Prefixes are allowed at the beginning of compounds, suffixes are allowed at the end of compounds by default. Affixes with COMPOUNDPERMITFLAG may be inside of compounds.



COMPOUNDFORBIDFLAG flag

Suffixes with this flag forbid compounding of the affixed word.



COMPOUNDROOT flag

COMPOUNDROOT flag signs the compounds in the dictionary (Now it is used only in the Hungarian language specific code).



COMPOUNDWORDMAX number



Set maximum word count in a compound word. (Default is unlimited.)



CHECKCOMPOUND DUP



Forbid word duplication in compounds (e.g. foofoo).



CHECKCOMPOUND REP



Forbid compounding, if the (usually bad) compound word may be a non compound word with a REP fault. Useful for languages with 'compound friendly' orthography.



CHECKCOMPOUND CASE



Forbid upper case characters at word bound in compounds.



CHECKCOMPOUND TRIPLE



Forbid compounding, if compound word contains triple letters (e.g. foo|ox or xo|oof). Bug: missing multi-byte character support in UTF-8 encoding (works only for 7-bit ASCII characters).



CHECKCOMPOUND PATTERN number_of_checkcompoundpattern_definitions



CHECKCOMPOUND PATTERN endchars beginchars



Forbid compounding, if first word in compound ends with endchars, and next word begins with beginchars.



COMPOUND SYLLABLE max_syllable vowels



Need for special compounding rules in Hungarian. First parameter is the maximum syllable number, that may be in a compound, if words in compounds are more than COMPOUNDWORDMAX. Second parameter is the list of vowels (for calculating syllables).



SYLLABLE NUM flags



Need for special compounding rules in Hungarian.



OPTIONS FOR AFFIX CREATION



PFX flag cross_product number



PFX flag stripping prefix condition [morphological_fields...]





SFX flag cross_product number





SFX flag stripping suffix condition [morphological_fields...]


An affix is either a prefix or a suffix attached to root words to make other words. We can define affix classes with arbitrary number affix rules. Affix classes are signed with affix flags. The first line of an affix class definition is the header. The fields of an affix class header:


 Option name (PFX or SFX)

 Flag (name of the affix class)


 Cross product (permission to combine prefixes and suffixes). Possible values: Y (yes) or N (no)


 Line count of the following rules.

 Fields of an affix rules:


 Option name


 Flag

 Stripping characters from beginning (at prefix rules) or end (at suffix rules) of the word

 Affix (optionally with flags of continuation classes, separated by a slash)

 Condition.

 Zero stripping or affix are indicated by zero. Zero condition is indicated by dot. Condition is a simplified, regular expression-like pattern, which must be met before the affix can be applied. (Dot signs an arbitrary character. Characters in braces sign an arbitrary character from the character subset. Dash hasn't got special meaning, but circumflex (^) next the first brace sets the complements character set.)

 Optional morphological fields separated by spaces or tabulators.



OTHER OPTIONS

CIRCUMFIX flag

Affixes signed with CIRCUMFIX flag may be on a word when this word also has a prefix with CIRCUMFIX flag and vice versa.



FORBIDDENWORD flag

This flag signs forbidden word form. Because affixed forms are also forbidden, we can subtract a subset from set of the accepted affixed and compound words.



KEEPCASE flag

Forbid uppercased and capitalized forms of words signed with KEEPCASE flags. Useful for special orthographies (measurements and currency often keep their case in uppercased texts) and writing systems (e.g. keeping lower case of IPA characters). Note: With CHECKSHARPS declaration, words with sharp s and KEEPCASE flag may be capitalized and uppercased, but uppercased forms of these words may not contain sharp s, only SS.



LEMMA_PRESENT flag

Not used in Hunspell 1.2. Use "st:" field instead of LEMMA_PRESENT.



NEEDAFFIX flag

This flag signs virtual stems in the dictionary. Only affixed forms of these words will be accepted by Hunspell. Except, if the dictionary word has a homonym or a zero affix. NEEDAFFIX works also with prefixes and prefix + suffix combinations (see tests/pseudoroot5.*).



PSEUDOROOT flag

Deprecated. (Former name of the NEEDAFFIX option.)



SUBSTANDARD flag

SUBSTANDARD flag signs affix rules and dictionary words (allomorphs) not used in morphological generation (and in suggestion in the future versions). See also NOSUGGEST.



WORDCHARS characters

WORDCHARS extends tokenizer of Hunspell command line interface with additional word character. For example, dot, dash, n-dash, numbers, percent sign are word character in Hungarian.



CHECKSHARPS

SS letter pair in uppercased (German) words may be upper case sharp s (ß). Hunspell can handle this special casing with the CHECKSHARPS declaration (see also KEEPCASE flag and tests/germancompounding example) in both



spelling and suggestion.



Morphological analysis

Hunspell's dictionary items and affix rules may have optional space or tabulator separated morphological description fields, started with 3-character (two letters and a colon) field IDs:



```
word/flags po:noun is:nom
```



Example: We define a simple resource with morphological informations, a derivative suffix (ds:) and a part of speech category (po:):



Test file:



```
SFX X Y 1
SFX X 0 able . ds:able
```



Dictionary file:



```
drink/X po:verb
```



Test file:



```
drink
drinkable
```



Test:



```
$ analyze test.aff test.dic test.txt
> drink
analyze(drink) = po:verb
stem(drink) = po:verb
> drinkable
analyze(drinkable) = po:verb ds:able
stem(drinkable) = drinkable
```



You can see in the example, that the analyzer concatenates the morphological fields in *item and arrangement* style.



Optional data fields

Default morphological and other IDs (used in suggestion, stemming and morphological generation):



ph: Alternative transliteration for better suggestion. It's useful for words with foreign pronunciation. (Dictionary based phonetic suggestion.) For example:



```
Marseille ph:maarsayl
```



st: Stem. Optional: default stem is the dictionary item in morphological analysis. Stem field is useful for virtual stems (dictionary words with NEEDAFFIX flag) and morphological exceptions instead of new, single used morphological rules.



```
feet st:foot is:plural
mice st:mouse is:plural
teeth st:tooth is:plural
```



Word forms with multiple stems need multiple dictionary items:

```
lay po:verb st:lie is:past_2
lay po:verb is:present
lay po:noun
```



al: Allomorph(s). A dictionary item is the stem of its allomorphs. Morphological generation needs stem, allomorph and affix fields.



```
sing al:sang al:sung
sang st:sing
sung st:sing
```



po: Part of speech category.

ds: Derivational suffix(es). Stemming doesn't remove derivational suffixes. Morphological generation depends on the order of the suffix fields.



In affix rules:

```
SFX Y Y 1
SFX Y 0 ly . ds:ly_adj
```



In the dictionary:

```
ably st:able ds:ly_adj
able al:ably
```



is: Inflectional suffix(es). All inflectional suffixes are removed by stemming. Morphological generation depends on the order of the suffix fields.

```
feet st:foot is:plural
```



ts: Terminal suffix(es). Terminal suffix fields are inflectional suffix fields "removed" by additional (not terminal) suffixes.

Useful for zero morphemes and affixes removed by splitting rules.

```
work/D ts:present
```



```
SFX D Y 2
```



```
SFX D 0 ed . is:past_1
SFX D 0 ed . is:past_2
```



Typical example of the terminal suffix is the zero morpheme of the nominative case.

- sp: Surface prefix. Temporary solution for adding prefixes to the stems and generated word forms. See tests/morph.* example.
- pa: Parts of the compound words. Output fields of morphological analysis for stemming.
- dp: Planned: derivational prefix.
- ip: Planned: inflectional prefix.
- tp: Planned: terminal prefix.



Twofold suffix stripping

Ispell's original algorithm strips only one suffix. Hunspell can strip another one yet (or a plus prefix in COMPLEXPREFIXES mode).



Twofold suffix stripping is a significant improvement in handling of immense number of suffixes, that characterize agglutinative languages.



Second 's' suffix (affix class Y) will be the continuation class of the suffix 'able' in the following example:



```
SFX Y Y 1
SFX Y 0 s .

SFX X Y 1
SFX X 0 able/Y .
```

Dictionary file:

```
drink/X
```

Test file:

```
drink
drinkable
drinkables
```

Test:

```
$ hunspell -m -d test <test.txt
drink st:drink
drinkable st:drink fl:X
drinkables st:drink fl:X fl:Y
```





Theoretically with the twofold suffix stripping needs only the square root of the number of suffix rules, compared with a Hunspell implementation. In our practice, we could have elaborated the Hungarian inflectional morphology with twofold suffix stripping.



Extended affix classes

Hunspell can handle more than 65000 affix classes. There are three new syntax for giving flags in affix and dictionary files.



FLAG long command sets 2-character flags:



```
FLAG long
SFX Y1 Y 1
SFX Y1 0 s 1
```



Dictionary record with the Y1, Z3, F? flags:



```
foo/Y1Z3F?
```



FLAG num command sets numerical flags separated by comma:



```
FLAG num
SFX 65000 Y 1
SFX 65000 0 s 1
```



Dictionary example:



```
foo/65000,12,2756
```



The third one is the Unicode character flags.



Homonyms

Hunspell's dictionary can contain repeating elements that are homonyms:



```
work/A    po:verb
work/B    po:noun
```



An affix file:



```
SFX A Y 1
SFX A 0 s . sf:sg3

SFX B Y 1
SFX B 0 s . is:plur
```



Test file:



```
works
```



Test:



```
$ hunspell -d test -m <testwords
work st:work po:verb is:sg3
work st:work po:noun is:plur
```



This feature also gives a way to forbid illegal prefix/suffix combinations.



Prefix--suffix dependencies

An interesting side-effect of multi-step stripping is, that the appropriate treatment of circumfixes now comes for free. For instance, in Hungarian, superlatives are formed by simultaneous prefixation of *leg-* and suffixation of *-bb* to the adjective base. A problem with the one-level architecture is that there is no way to render lexical licensing of particular prefixes and suffixes interdependent, and therefore incorrect forms are recognized as valid, i.e. **legvén = leg + vén* ‘old’. Until the introduction of clusters, a special treatment of the superlative had to be hardwired in the earlier **HunSpell** code. This may have been legitimate for a single case, but in fact prefix--suffix dependences are ubiquitous in category-changing derivational patterns (cf. English *payable, non-payable* but **non-pay* or *drinkable, undrinkable* but **undrink*). In simple words, here, the prefix *un-* is legitimate only if the base *drink* is suffixed with *-able*. If both these patterns are handled by on-line affix rules and affix rules are checked against the base only, there is no way to express this dependency and the system will necessarily over- or undergenerate.



Next example, suffix class R have got a prefix ‘continuation’ class (class P).



```
PFX P Y 1
PFX P 0 un . [prefix_un]+

SFX S Y 1
SFX S 0 s . +PL

SFX Q Y 1
SFX Q 0 s . +3SGV

SFX R Y 1
SFX R 0 able/PS . +DER_V_ADJ_ABLE
```



Dictionary:



```
2
drink/      [verb]
RQ
```



```
drink/      [noun]
S
```



Morphological analysis:



```
> drink
drink[verb]
```



```

drink[noun]
> drinks
drink[verb]+3SGV
drink[noun]+PL
> drinkable
drink[verb]+DER_V_ADJ_ABLE
> drinkables
drink[verb]+DER_V_ADJ_ABLE+PL
> undrinkable
[prefix_un]+drink[verb]+DER_V_ADJ_ABLE
> undrinkables
[prefix_un]+drink[verb]+DER_V_ADJ_ABLE+PL
> undrink
Unknown word.
> undrinks
Unknown word.

```



Circumfix

Conditional affixes implemented by a continuation class are not enough for circumfixes, because a circumfix is one affix in morphology. We also need CIRCUMFIX option for correct morphological analysis.



```

# circumfixes: ~ obligate prefix/suffix combinations
# superlative in Hungarian: leg- (prefix) AND -bb (suffix)
# nagy, nagyobb, legnagyobb, legeslegnagyobb
# (great, greater, greatest, most greatest)

```

```
CIRCUMFIX X
```

```
PFX A Y 1
PFX A 0 leg/X .
```

```
PFX B Y 1
PFX B 0 legesleg/X .
```

```
SFX C Y 3
SFX C 0 obb . +COMPARATIVE
SFX C 0 obb/AX . +SUPERLATIVE
SFX C 0 obb/BX . +SUPERSUPERLATIVE

```



Dictionary:



```
1
```



```
nagy/      [MN]
C
```



Analysis:



```

> nagy
nagy[MN]
> nagyobb
nagy[MN]+COMPARATIVE
> legnagyobb
nagy[MN]+SUPERLATIVE

```

```
> legeslegnagyobb
nagy[MN]+SUPERSUPERLATIVE
```



Compounds

Allowing free compounding yields decrease in precision of recognition, not to mention stemming and morphological analysis. Although lexical switches are introduced to license compounding of bases by **Ispell**, this proves not to be restrictive enough. For example:



```
# affix file
COMPOUNDFLAG X
```



```
2
foo/X
bar/X
```



With this resource, *foobar* and *barfoo* also are accepted words.



has been improved upon with the introduction of direction-sensitive compounding, i.e., lexical features can specify separately whether a base can occur as leftmost or rightmost constituent in compounds. This, however, is still insufficient to handle the intricate patterns of compounding, not to mention idiosyncratic (and language specific) norms of hyphenation.



Hunspell algorithm currently allows any affixed form of words, which are lexically marked as potential members of compounds. **Hunspell** improved this, and its recursive compound checking rules makes it possible to implement the intricate spelling conventions of Hungarian compounds. For example, using **COMPOUNDWORDMAX**, **COMPOUNDSYLLABLE**, **COMPOUNDROOT**, **SYLLABLENUM** options can be set the noteworthy Hungarian ‘6-3’ rule. Further example in Hungarian, derivate suffixes often modify compounding properties. Hunspell allows the compounding flags on the affixes, and there are two special flags (**COMPOUNDPERMITFLAG** and **COMPOUNDFORBIDFLAG**) to permit or prohibit compounding of the derivations.



Prefixes with this flag forbid compounding of the affixed word.



also need several Hunspell features for handling German compounding:

```
# German compounding

# set language to handle special casing of German sharp s
LANG de_DE

# compound flags

COMPOUNDBEGIN U
COMPOUNDMIDDLE V
COMPOUNDEND W

# Prefixes are allowed at the beginning of compounds,
# suffixes are allowed at the end of compounds by default:
# (prefix)?(root)+(affix)?
```

```

# Affixes with COMPOUNDPERMITFLAG may be inside of compounds.
COMPOUNDPERMITFLAG P

# for German fogemorphemes (Fuge-element)
# Hint: ONLYINCOMPOUND is not required everywhere, but the
# checking will be a little faster with it.

ONLYINCOMPOUND X

# forbid uppercase characters at compound word bounds
CHECKCOMPOUNDCASE

# for handling Fuge-elements with dashes (Arbeits-)
# dash will be a special word

COMPOUNDMIN 1
WORDCHARS -

# compound settings and fogemorpheme for 'Arbeit'

SFX A Y 3
SFX A 0 s/UPX .
SFX A 0 s/VPDX .
SFX A 0 0/WXD .

SFX B Y 2
SFX B 0 0/UPX .
SFX B 0 0/VWXDP .

# a suffix for 'Computer'

SFX C Y 1
SFX C 0 n/WD .

# for forbid exceptions (*Arbeitsnehmer)

FORBIDDENWORD Z

# dash prefix for compounds with dash (Arbeits-Computer)

PFX - Y 1
PFX - 0 -/P .

# decapitalizing prefix
# circumfix for positioning in compounds

PFX D Y 29
PFX D A a/PX A
PFX D Ä ä/PX Ä
.
.
PFX D Y y/PX Y
PFX D Z z/PX Z

```



Example dictionary:

```

4
Arbeit/A-
Computer/BC-

```

-/W
Arbeitsnehmer/Z



Accepted compound compound words with the previous resource:

Computer
Computern
Arbeit
Arbeits-
Computerarbeit
Computerarbeits-
Arbeitscomputer
Arbeitscomputern
Computerarbeitscomputer
Computerarbeitscomputern
Arbeitscomputerarbeit
Computerarbeits-Computer
Computerarbeits-Computern



Not accepted compoundings:

computer
arbeit
Arbeits
arbeits
ComputerArbeit
ComputerArbeits
Arbeitcomputer
ArbeitsComputer
Computerarbeitcomputer
ComputerArbeitcomputer
ComputerArbeitscomputer
Arbeitscomputerarbeits
Computerarbeits-computer
Arbeitsnehmer



This solution is still not ideal, however, and will be replaced by a pattern-based compound-checking algorithm which is closely integrated with input buffer tokenization. Patterns describing compounds come as a separate input resource that can refer to high-level properties of constituent parts (e.g. the number of syllables, affix flags, and containment of hyphens). The patterns are matched against potential segmentations of compounds to assess wellformedness.



Unicode character encoding

Both **Ispell** and **Myspell** use 8-bit ASCII character encoding, which is a major deficiency when it comes to scalability. Although a language like Hungarian has a standard ASCII character set (ISO 8859-2), it fails to allow a full implementation of Hungarian orthographic conventions. For instance, the '–' symbol (n-dash) is missing from this character set contrary to the fact that it is not only the official symbol to delimit parenthetic clauses in the language, but it can be in compound words as a special 'big' hyphen. **Ispell** has got some 8-bit encoding tables, but there are languages without standard 8-bit encoding, too. For example, a lot of African languages have non-latin or extended latin characters.



Similarly, using the original spelling of certain foreign names like *Ångström* or *Molière* is encouraged by the Hungarian spelling norm, and, since characters 'Å' and 'è' are not part of ISO 8859-2, when they combine with inflections containing characters only in ISO 8859-2 (like elative *-bl*, allative *-tl* or delative *-rl* with double acute), these result in words (like *Ångströmrl* or *Molière-tl.*) that can not be encoded using any single ASCII encoding scheme.



Problems raised in relation to 8-bit ASCII encoding have long been recognized by proponents of Unicode. It is clear that trading efficiency for encoding-independence has its advantages when it comes a truly multi-lingual application. There is implemented a memory and time efficient Unicode handling in Hunspell. In non-UTF-8 character encodings Hunspell works with the original 8-bit strings. In UTF-8 encoding, affixes and words are stored in UTF-8, during the analysis are handled in mostly UTF-8, under condition checking and suggestion are converted to UTF-16. Unicode text analysis and spell checking have a minimal (0-20%) time overhead and minimal or reasonable memory overhead depends from the language (its UTF-8 encoding and affixation).



SEE ALSO

hunspell (1), ispell (1), ispell (4)

Hunspell API



NAME

hunspell - spell checking, stemming, morphological generation and analysis



SYNOPSIS

```
#include <hunspell/hunspell.hxx> /* or */
#include <hunspell/hunspell.h>

hunspell(const char *affpath, const char *dpath);
hunspell(const char *affpath, const char *dpath, const char *key);
hunspell();
add_dic(const char *dpath);
add_dic(const char *dpath, const char *key);
spell(const char *word);
spell(const char *word, int *info, char **root);
suggest(char***slst, const char *word);
analyze(char***slst, const char *word);
stem(char***slst, const char *word);
stem(char***slst, char **morph, int n);
generate(char***slst, const char *word, const char *word2);
generate(char***slst, const char *word, char **desc, int n);
free_list(char ***slst, int n);
add(const char *word);
add_with_affix(const char *word, const char *example);
remove(const char *word);
char * get_dic_encoding();
const char * get_wordchars();
unsigned short * get_wordchars_utf16(int *len);
const cs_info * get_csconv();
const char * get_version();
```



DESCRIPTION

The **Hunspell** library routines give the user word-level linguistic functions: spell checking and correction, stemming, morphological generation and analysis in item-and-arrangement style.



optional C header contains the C interface of the C++ library with `Hunspell_create` and `Hunspell_destroy` constructor and destructor, and an extra `HunHandle` parameter (the allocated object) in the wrapper functions (see in the C header file **hunspell.h**).



basic spelling functions, **spell()** and **suggest()** can be used for stemming, morphological generation and analysis by XML input texts (see XML API).



Constructor and destructor

Hunspell's constructor needs paths of the affix and dictionary files. See the **hunspell(4)** manual page for the dictionary format. Optional **key** parameter is for dictionaries encrypted by the **hzip** tool of the Hunspell distribution.




Extra dictionaries

The `add_dic()` function load an extra dictionary file. The extra dictionaries use the affix file of the allocated Hunspell object. Maximal number of the extra dictionaries is limited in the source code (20).



Spelling and correction

The `spell()` function returns non-zero, if the input word is recognised by the spell checker, and a zero value if not. Optional reference variables return a bit array (info) and the root word of the input word. Info bits checked with the `SPELL_COMPOUND` and `SPELL_FORBIDDEN` macros sign compound words and explicit forbidden words.  `suggest()` function has two input parameters, a reference variable of the output suggestion list, and an input word. The function returns the number of the suggestions. The reference variable will contain the address of the newly allocated suggestion list or `NULL`, if the return value of `suggest()` is zero. Maximal number of the suggestions is limited in the source code.



`spell()` and `suggest()` can recognize XML input, see the XML API section.



Morphological functions

The plain `stem()` and `analyze()` functions are similar to the `suggest()`, but instead of suggestions, return stems and results of the morphological analysis. The plain `generate()` waits a second word, too. This extra word and its affixation will be the model of the morphological generation of the requested forms of the first word.



extended `stem()` and `generate()` use the results of a morphological analysis:



```
char ** result, result2;
int n1 = analyze(&result, "words");
```



```
int n2 = stem(&result2, result, n1);
```

The morphological annotation of the Hunspell library has fixed (two letter and a colon) field identifiers, see the **hunspell(4)** manual page.

```
char ** result;  
char * affix = "is:plural"; // description depends from dictionaries, too  
int n = generate(&result, "word", &affix, 1);  
for (int i = 0; i < n; i++) printf("%s0", result[i]);
```

Memory deallocation

The `free_list()` function frees the memory allocated by `suggest()`, `analyze`, `generate` and `stem()` functions.

Other functions

The `add()`, `add_with_affix()` and `remove()` are helper functions of a personal dictionary implementation to add and remove words from the base dictionary in run-time. The `add_with_affix()` uses a second word as a model of the enabled affixation of the new word.

`get_dic_encoding()` function returns "ISO8859-1" or the character encoding defined in the affix file with the "SET" keyword.

`get_cconv()` function returns the 8-bit character case table of the encoding of the dictionary.

`get_wordchars()` and `get_wordchars_utf16()` return the extra word characters defined in affix file for tokenization by the "WORDCHARS" keyword.

`get_version()` returns the version string of the library.

XML API

The `spell()` function returns non-zero for the "<?xml?>" input indicating the XML API support. `suggest()` function stems, analyzes and generates the forms of the input word, if it was added by one of the following "SPELLML" syntaxes:

```
<?xml?>  
<query type="analyze">  
<word>dogs</word>  
</query>
```

```
<?xml?>  
<query type="stem">
```



```
<word>dogs</word>
</query>
```



```
<?xml?>
<query type="generate">
<word>dog</word>
<word>cats</word>
</query>
```



```
<?xml?>
<query type="generate">
<word>dog</word>
<code><a>is:pl</a><a>is:poss</a></code>
</query>
```

The outputs of the type="stem" query and the stem() library function are the same. The output of the type="analyze" query is a string contained a `<a>result1<a>result2...` element. This element can be used in the second syntax of the type="generate" query.



EXAMPLE

See analyze.cxx in the Hunspell distribution.



AUTHORS

Hunspell based on Ispell's spell checking algorithms and OpenOffice.org's Myspell source code.

Author of International Ispell is Geoff Kuenning. Author of MySpell is Kevin Hendricks. Author of Hunspell is László Németh. Author of the original C API is Caolan McNamara. Author of the Aspell table-driven phonetic transcription algorithm and code is Björn Jacke. Also THANKS and Changelog files of Hunspell distribution.