NAME

ispell − format of ispell dictionaries and affix files

DESCRIPTION

Ispell(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. The two files are combined by *buildhash* (see *ispell*(1)) and written to a hash file which is not described here.

A raw *ispell* dictionary (either the main dictionary or your own personal dictionary) contains a list of words, one per line. Each word may optionally be followed by a slash ("/") and one or more flags, which modify the root word as explained below. Depending on the options with which *ispell* was built, case may or may not be significant in either the root word or the flags, independently. Specifically, if the compile-time option CAPITALIZATION is defined, case is significant in the root word; if not, case is ignored in the root word. If the compile-time option MASKBITS is set to a value of 32, case is ignored in the flags; otherwise case is significant in the flags. Contact your system administrator or *ispell* maintainer for more information (or use the **−vv** flag to find out). The dictionary should be sorted with the **−f** flag of *sort*(1) before the hash file is built; this is done automatically by *munchlist*(1), which is the normal way of building dictionaries.

If the dictionary contains words that have string characters (see the affix-file documentation below), they must be written in the format given by the **defstringtype** statement in the affix file. This will be the case for most non-English languages. Be careful to use this format, rather than that of your favorite formatter, when adding words to a dictionary. (If you add words to your personal dictionary during an *ispell* session, they will automatically be converted to the correct format. This feature can be used to convert an entire dictionary if necessary:)

```
echo qqqqq > dummy.dict
buildhash dummy.dict affix-file dummy.hash
awk '{print "*"}END{print "#"}' old-dict-file \
| ispell -a -T old-dict-string-type \
  -d ./dummy.hash -p ./new-dict-file \
  > /dev/null
rm dummy.*
```

The case of the root word controls the case of words accepted by *ispell*, as follows:

(1) If the root word appears only in lower case (e.g., *bob*), it will be accepted in lower case, capitalized, or all capitals.

(2) If the root word appears capitalized (e.g., *Robert*), it will not be accepted in all-lower case, but will be accepted capitalized or all in capitals.

(3) If the root word appears all in capitals (e.g., *UNIX*), it will only be accepted all in capitals.

(4) If the root word appears with a "funny" capitalization (e.g., *ITCorp*), a word will be accepted only if it follows that capitalization, or if it appears all in capitals.

(5) More than one capitalization of a root word may appear in the dictionary. Flags from different capitalizations are combined by OR-ing them together.

Redundant capitalizations (e.g., *bob* and *Bob*) will be combined by *buildhash* and by *ispell* (for personal dictionaries), and can be removed from a raw dictionary by *munchlist*.

For example, the dictionary:

```
bob
Robert
UNIX
ITcorp
ITCorp
```

will accept *bob*, *Bob*, *BOB*, *Robert*, *ROBERT*, *UNIX*, *ITcorp*, *ITCorp*, and *ITCORP*, and will reject all

others. Some of the unacceptable forms are *bOb*, *robert*, *Unix*, and *ItCorp*.

As mentioned above, root words in any dictionary may be extended by flags. Each flag is a single alphabetic character, which represents a prefix or suffix that may be added to the root to form a new word. For example, in an English dictionary the **D** flag can be added to *bathe* to make *bathed*. Since flags are represented as a single bit in the hashed dictionary, this results in significant space savings. The *munchlist* script will reduce an existing raw dictionary by adding flags when possible.

When a word is extended with an affix, the affix will be accepted only if it appears in the same case as the initial (prefix) or final (suffix) letter of the word. Thus, for example, the entry *UNIX/M* in the main dictionary (**M** means add an apostrophe and an "s" to make a possessive) would accept *UNIX'S* but would reject *UNIX's*. If *UNIX's* is legal, it must appear as a separate dictionary entry, and it will not be combined by *munchlist*. (In general, you don't need to worry about these things; *munchlist* guarantees that its output dictionary will accept the same set of words as its input, so all you have to do is add words to the dictionary and occasionally run munchlist to reduce its size).

As mentioned, the affix definition file describes the affixes associated with particular flags. It also describes the character set used by the language.

Although the affix-definition grammar is designed for a line-oriented layout, it is actually a free-format yacc grammar and can be laid out weirdly if you want. Comments are started by a pound (sharp) sign (#), and continue to the end of the line. Backslashes are supported in the usual fashion (*nnn*, plus specials **\n**, **\r**, **\t**, **\v**, **\f**, **\b**, and the new hex format **\x***nn*). Any character with special meaning to the parser can be changed to an uninterpreted token by backslashing it; for example, you can declare a flag named 'asterisk' or 'colon' with *flag *:* or *flag \::*.

The grammar will be presented in a top-down fashion, with discussion of each element. An affix-definition file must contain exactly one table:

table : [*headers*] [*prefixes*] [*suffixes*]

At least one of *prefixes* and *suffixes* is required. They can appear in either order.

headers : [*options*] *char-sets*

The headers describe options global to this dictionary and language. These include the character sets to be used and the formatter, and the defaults for certain *ispell* flags.

options : { *fmtr-stmt* | *opt-stmt* | *flag-stmt* | *num-stmt* }

The options statements define the defaults for certain ispell flags and for the character sets used by the formatters.

fmtr-stmt : { *nroff-stmt* | *tex-stmt* }

A *fmtr-stmt* describes characters that have special meaning to a formatter. Normally, this statement is not necessary, but some languages may have preempted the usual defaults for use as language-specific characters. In this case, these statements may be used to redefine the special characters expected by the formatter.

nroff-stmt : { **nroffchars** | **troffchars** } *string*

The **nroffchars** statement allows redefinition of certain *nroff* control characters. The string given must be exactly five characters long, and must list substitutions for the left and right parentheses ("()") , the period ("."), the backslash ("\"), and the asterisk ("*"). (The right parenthesis is not currently used, but is included for completeness.) For example, the statement:

nroffchars {}.*

would replace the left and right parentheses with left and right curly braces for purposes of parsing *nroff*/*troff* strings, with no effect on the others (admittedly a contrived example). Note that the backslash is escaped with a backslash.

tex-stmt : { **TeXchars** | **texchars** } *string*

The **TeXchars** statement allows redefinition of certain TeX/LaTeX control characters. The string given must be exactly thirteen characters long, and must list substitutions for the left and right parentheses ("()") ,

the left and right square brackets ("[]"), the left and right curly braces ("{}"), the left and right angle brackets ("<>"), the backslash ("\"), the dollar sign ("$"), the asterisk ("*"), the period or dot ("."), and the percent sign ("%"). For example, the statement:

texchars ()\[ ]<\><\>\\$*.%

would replace the functions of the left and right curly braces with the left and right angle brackets for purposes of parsing TeX/LaTeX constructs, while retaining their functions for the *tib* bibliographic preprocessor. Note that the backslash, the left square bracket, and the right angle bracket must be escaped with a backslash.

opt-stmt : { *cmpnd-stmt* | *aff-stmt* }

cmpnd-stmt : **compoundwords** *compound-opt*

aff-stmt : **allaffixes** *on-or-off*

on-or-off: { **on** | **off** }

compound-opt : { *on-or-off* | **controlled** *character* }

An *opt-stmt* controls certain ispell defaults that are best made language-specific. The **allaffixes** statement controls the default for the **−P** and **−m** options to *ispell.* If **allaffixes** is turned **off** (the default), *ispell* will default to the behavior of the *−P* flag: root/affix suggestions will only be made if there are no "near misses". If **allaffixes** is turned **on**, *ispell* will default to the behavior of the *−m* flag: root/affix suggestions will always be made. The **compoundwords** statement controls the default for the **−B** and **−C** options to *ispell.* If **compoundwords** is turned **off** (the default), *ispell* will default to the behavior of the *−B* flag: run-together words will be reported as errors. If **compoundwords** is turned **on**, *ispell* will default to the behavior of the *−C* flag: run-together words will be considered as compounds if both are in the dictionary. This is useful for languages such as German and Norwegian, which form large numbers of compound words. Finally, if **compoundwords** is set to *controlled*, only words marked with the flag indicated by *character* (which should not be otherwise used) will be allowed to participate in compound formation. Because this option requires the flags to be specified in the dictionary, it is not available from the command line.

flag-stmt: **flagmarker** *character*

The **flagmarker** statement describes the character which is used to separate affix flags from the root word in a raw dictionary file. This must be a character which is not found in any word (including in string characters; see below). The default is "/" because this character is not normally used to represent special characters in any language.

num-stmt : **compoundmin** *digit*

The **compoundmin** statement controls the length of the two components of a compound word. This only has an effect if **compoundwords** is turned **on** or if the **−C** flag is given to *ispell*. In that case, only words at least as long as the given minimum will be accepted as components of a compound. The default is 3 characters.

char-sets : *norm-sets* [*alt-sets*]

The character-set section describes the characters that can be part of a word, and defines their collating order. There must always be a definition of "normal" character sets; in addition, there may be one or more partial definitions of "alternate" sets which are used with various text formatters.

norm-sets : [*deftype*] charset-group

A "normal" character set may optionally begin with a definition of the file suffixes that make use of this set. Following this are one or more character-set declarations.

deftype : **defstringtype** *name deformatter suffix**

The **defstringtype** declaration gives a list of file suffixes which should make use of the default string characters defined as part of the base character set; it is only necessary if string characters are being defined.

The *name* parameter is a string giving the unique name associated with these suffixes; often it is a formatter name. If the formatter is a member of the troff family, "nroff" should be used for the name associated with the most popular macro package; members of the TeX family should use "tex". Other names may be chosen freely, but they should be kept simple, as they are used in *ispell 's* **−T** switch to specify a formatter type. The *deformatter* parameter specifies the deformatting style to use when processing files with the given suffixes. Currently, this must be either **tex** or **nroff**. The *suffix* parameters are a whitespace-separated list of strings which, if present at the end of a filename, indicate that the associated set of string characters should be used by default for this file. For example, the suffix list for the troff family typically includes suffixes such as ".ms", ".me", ".mm", etc.

charset-group : { *char-stmt* | *string-stmt* | *dup-stmt*}*

A *char-stmt* describes single characters; a *string-stmt* describes characters that must appear together as a string, and which usually represent a single character in the target language. Either may also describe conversion between upper and lower case. A *dup-stmt* is used to describe alternate forms of string characters, so that a single dictionary may be used with several formatting programs that use different conventions for representing non-ASCII characters.

char-stmt	:	**wordchars** *character-range*
	|	**wordchars** *lowercase-range uppercase-range*
	|	**boundarychars** *character-range*
	|	**boundarychars** *lowercase-range uppercase-range*
string-stmt	:	**stringchar** *string*
	|	**stringchar** *lowercase-string uppercase-string*

Characters described with the **boundarychars** statement are considered part of a word only if they appear singly, embedded between characters declared with the **wordchars** or **stringchar** statements. For example, if the hyphen is a boundary character (useful in French), the string "foo-bar" would be a single word, but "-foo" would be the same as "foo", and "foo--bar" would be two words separated by non-word characters.

If two ranges or strings are given in a *char-stmt* or *string-stmt*, the first describes characters that are interpreted as lowercase and the second describes uppercase. In the case of a **stringchar** statement, the two strings must be of the same length. Also, in a **stringchar** statement, the actual strings may contain both uppercase and characters themselves without difficulty; for instance, the statement

stringchar "*(sS" "*(Ss"

is legal and will not interfere with (or be interfered with by) other declarations of of "s" and "S" as lower and upper case, respectively.

A final note on string characters: some languages collate certain special characters as if they were strings. For example, the German "a-umlaut" is traditionally sorted as if it were "ae". Ispell is not capable of this; each character must be treated as an individual entity. So in certain cases, ispell will sort a list of words into a different order than the standard "dictionary" order for the target language.

alt-sets : *alttype* [*alt-stmt**]

Because different formatters use different notations to represent non-ASCII characters, *ispell* must be aware of the representations used by these formatters. These are declared as alternate sets of string characters.

alttype : **altstringtype** *name suffix**

The **altstringtype** statement introduces each set by declaring the associated formatter name and filename suffix list. This name and list are interpreted exactly as in the **defstringtype** statement above. Following this header are one or more *alt-stmt*s which declare the alternate string characters used by this formatter.

alt-stmt : **altstringchar** *alt-string std-string*

The *altstringchar* statement describes alternate representations for string characters. For example, the −mm macro package of *troff* represents the German "a-umlaut" as *a*:*, while *TeX* uses the sequence *\"a*. If the *troff* versions are declared as the standard versions using **stringchar**, the *TeX* versions may be declared as alternates by using the statement

altstringchar \\\"a a*:

When the **altstringchar** statement is used to specify alternate forms, all forms for a particular formatter must be declared together as a group. Also, each formatter or macro package must provide a complete set of characters, both upper- and lower-case, and the character sequences used for each formatter must be completely distinct. Character sequences which describe upper- and lower-case versions of the same printable character must also be the same length. It may be necessary to define some new macros for a given formatter to satisfy these restrictions. (The current version of *buildhash* does not enforce these restrictions, but failure to obey them may result in errors being introduced into files that are processed with *ispell*.)

An important minor point is that *ispell* assumes that all characters declared as **wordchars** or **boundarychars** will occupy exactly one position on the terminal screen.

A single character-set statement can declare either a single character or a contiguous range of characters. A range is given as in egrep and the shell: [a-z] means lowercase alphabetics; [^a-z] means all but lowercase, etc. All character-set statements are combined (unioned) to produce the final list of characters that may be part of a word. The collating order of the characters is defined by the order of their declaration; if a range is used, the characters are considered to have been declared in ASCII order. Characters that have case are collated next to each other, with the uppercase character first.

The character-declaration statements have a rather strange behavior caused by its need to match each lowercase character with its uppercase equivalent. In any given **wordchars** or **boundarychars** statement, the characters in each range are first sorted into ASCII collating sequence, then matched one-for-one with the other range. (The two ranges must have the same number of characters). Thus, for example, the two statements:

wordchars [aeiou] [AEIOU]
wordchars [aeiou] [UOIEA]

would produce exactly the same effect. To get the vowels to match up "wrong", you would have to use separate statements:

wordchars a U
wordchars e O
wordchars i I
wordchars o E
wordchars u A

which would cause uppercase 'e' to be 'O', and lowercase 'O' to be 'e'. This should normally be a problem only with languages which have been forced to use a strange ASCII collating sequence. If your uppercase and lowercase letters both collate in the same order, you shouldn't have to worry about this "feature".

The prefixes and suffixes sections have exactly the same syntax, except for the introductory keyword.

prefixes : **prefixes** *flagdef**
suffixes : **suffixes** *flagdef**
flagdef : **flag** [*****|~] *char* **:** *repl**

A prefix or suffix table consists of an introductory keyword and a list of flag definitions. Flags can be defined more than once, in which case the definitions are combined. Each flag controls one or more *repl*s (replacements) which are conditionally applied to the beginnings or endings of various words.

Flags are named by a single character *char*. Depending on a configuration option, this character can be either any uppercase letter (the default configuration) or any 7-bit ASCII character. Most languages should be able to get along with just 26 flags.

A flag character may be prefixed with one or more option characters. (If you wish to use one of the option characters as a flag character, simply enclose it in double quotes.)

The asterisk (*****) option means that this flag participates in *cross-product* formation. This only matters if the file contains both prefix and suffix tables. If so, all prefixes and suffixes marked with an asterisk will be applied in all cross-combinations to the root word. For example, consider the root *fix* with prefixes *pre* and *in*, and suffixes *es* and *ed*. If all flags controlling these prefixes and suffixes are marked with an asterisk,

then the single root *fix* would also generate *prefix*, *prefixes*, *prefixed*, *infix*, *infixes*, *infixed*, *fix*, *fixes*, and *fixed*. Cross-product formation can produce a large number of words quickly, some of which may be illegal, so watch out. If cross-products produce illegal words, *munchlist* will not produce those flag combinations, and the flag will not be useful.

repl : *condition** **>** [**-** *strip-string* **,**] *append-string*

The **~** option specifies that the associated flag is only active when a compound word is being formed. This is useful in a language like German, where the form of a word sometimes changes inside a compound.

A *repl* is a conditional rule for modifying a root word. Up to 8 *conditions* may be specified. If the *conditions* are satisfied, the rules on the right-hand side of the *repl* are applied, as follows:

(1) If a strip-string is given, it is first stripped from the beginning or ending (as appropriate) of the root word.

(2) Then the append-string is added at that point.

For example, the *condition* **.** means "any word", and the *condition* **Y** means "any word ending in Y". The following (suffix) replacements:

```
.       >       MENT
Y       >       -Y,IES
```

would change *induce* to *inducement* and *fly* to *flies*. (If they were controlled by the same flag, they would also change *fly* to *flyment*, which might not be what was wanted. *Munchlist* can be used to protect against this sort of problem; see the command sequence given below.)

No matter how much you might wish it, the strings on the right must be strings of specific characters, not ranges. The reasons are rooted deeply in the way *ispell* works, and it would be difficult or impossible to provide for more flexibility. For example, you might wish to write:

```
[EY]    >       -[EY],IES
```

This will not work. Instead, you must use two separate rules:

```
E       >       -E,IES
Y       >       -Y,IES
```

The application of *repl*s can be restricted to certain words with *conditions*:

condition : { **.** | *character* | *range* }

A *condition* is a restriction on the characters that adjoin, and/or are replaced by, the right-hand side of the *repl*. Up to 8 *conditions* may be given, which should be enough context for anyone. The right-hand side will be applied only if the *conditions* in the *repl* are satisfied. The *conditions* also implicitly define a length; roots shorter than the number of *conditions* will not pass the test. (As a special case, a *condition* of a single dot "." defines a length of zero, so that the rule applies to all words indiscriminately). This length is independent of the separate test that insists that all flags produce an output word length of at least four.

Conditions that are single characters should be separated by white space. For example, to specify words ending in "ED", write:

```
E D     >       -ED,ING                 # As in covered > covering
```

If you write:

```
ED      >       -ED,ING
```

the effect will be the same as:

```
[ED]    >       -ED,ING
```

As a final minor, but important point, it is sometimes useful to rebuild a dictionary file using an incompatible suffix file. For example, suppose you expanded the "R" flag to generate "er" and "ers" (thus making the Z flag somewhat obsolete). To build a new dictionary *newdict* that, using *newaffixes*, will accept exactly the same list of words as the old list *olddict* did using *oldaffixes*, the **−c** switch of *munchlist* is useful, as in the following example:

```
$ munchlist -c oldaffixes -l newaffixes olddict > newdict
```

If you use this procedure, your new dictionary will always accept the same list the original did, even if you badly screwed up the affix file. This is because *munchlist* compares the words generated by a flag with the original word list, and refuses to use any flags that generate illegal words. (But don't forget that the *munchlist* step takes a long time and eats up temporary file space).

EXAMPLES

As an example of conditional suffixes, here is the specification of the **S** flag from the English affix file:

```
flag *S:
  [^AEIOU]Y     >       -Y,IES  # As in imply > implies
  [AEIOU]Y      >       S               # As in convey > conveys
  [SXZH]        >       ES              # As in fix > fixes
  [^SXZHY]      >       S               # As in bat > bats
```

The first line applies to words ending in Y, but not in vowel-Y. The second takes care of the vowel-Y words. The third then handles those words that end in a sibilant or near-sibilant, and the last picks up everything else.

Note that the *conditions* are written very carefully so that they apply to disjoint sets of words. In particular, note that the fourth line excludes words ending in Y as well as the obvious SXZH. Otherwise, it would convert "imply" into "implys".

Although the English affix file does not do so, you can also have a flag generate more than one variation on a root word. For example, we could extend the English "R" flag as follows:

```
flag *R:
  E             >       R               # As in skate > skater
  E             >       RS              # As in skate > skaters
  [^AEIOU]Y     >       -Y,IER  # As in multiply > multiplier
  [^AEIOU]Y     >       -Y,IERS # As in multiply > multipliers
  [AEIOU]Y      >       ER              # As in convey > conveyer
  [AEIOU]Y      >       ERS             # As in convey > conveyers
  [^EY]         >       ER              # As in build > builder
  [^EY]         >       ERS             # As in build > builders
```

This flag would generate both "skater" and "skaters" from "skate". This capability can be very useful in languages that make use of noun, verb, and adjective endings. For instance, one could define a single flag that generated all of the German "weak" verb endings.

SEE ALSO

ispell(1)