# XƎSearch

v.0.1

2009/11/04

Paul Isambert

zappathustra@free.fr

## Contents

# 0 Changes

**2009/11/04** Corrected for ConTEXt (thanks to Wolfgang Schuster):
Now there's a third party file, `t-XƎSearch.tex`, so that `XƎSearch` can be properly loaded with `\usemodule[XƎSearch]`.
The clash between ConTEXt's `\unexpanded` macro and XƎTEX's (actually $\varepsilon$-TEX's) `\unexpanded` primitive has been fixed.

**2009/10/24** Initial version

# 1 Introductory remarks

1. This set of macros requires the XƎTEX engine.
2. This set of macros is totally experimental.
3. This set of macros is written with plain XƎTEX, and so it should be compatible with all formats, at least if they implement such basic macros as `\newcount` or `\newif`, which is the case at least for LATEX and ConTEXt.
4. As a consequence of the preceding remark, I've used in the examples of this documentation control sequences that don't exist in any format (as far as I know) but whose meaning is transparent enough, like `\blue` or `\italics`, which typeset blue and *italics*. They are not part of XƎSearch.
5. This set of macros tweaks XƎTEX's character class mechanism badly. This mechanism was not designed to do what it does here. Anyway, since it is used mainly for non-alphabetical writing systems, there's little chance of clashing with XƎSearch. I have tried to make XƎSearch compatible with François Charette's polyglossia for language with special punctuation pattern, like French. I have not tried to patch babel German shorthands in polyglossia, simply because I was not able to make them work.
6. XƎSearch is local all the way down, that is, there's not a single global command. So it can be used in a controlled way.[1]
7. To see what XƎSearch does, see **example 1** on the right.
8. To load the package in LATEX, say

```
\usepackage{xesearch}
```

In ConTEXt:

```
\usemodule[xesearch]
```

In plain XƎTEX:

```
\input xesearch.sty
```

[1]If your knowledge of TEX is confined to LATEX, you might not be very familiar with the notion of locality to groups, since in LATEX pretty much everything is global by default, whereas in plain TEX the contrary holds. So to make things simple, just remember that if you use XƎSearch inside a LATEX environment, even one you've defined yourself with `\newenvironment`, nothing will spread outside this environment. (I don't know the situation for ConTEXt, so I won't say anything.)

```
\SearchList{color}{\csname#1\endcsname{#1}}{blue,red,green}
This is blue and this is red and this is green,
but apparently yellow was not defined.
```

This is blue and this is red and this is green, but apparently yellow was not defined.

**Example 1: A Simple Example**

## 2 Let's search

- **\SearchList**⟨**!*⟩{⟨***name***⟩}{⟨***replacement text***⟩}{⟨***list of words***⟩}

The star and exclamation mark are optional and their relative order does not matter. Sticking to mandatory arguments for the moment, here's how this macro works: first, you give a ⟨*name*⟩ to this list, for further reference. Then you specify the ⟨*replacement text*⟩, which will be substituted for all of the words in ⟨*list of words*⟩ (separated by commas). In this ⟨*replacement text*⟩, the substituted word is designed by `#1`, so just think about it as an argument to a control sequence. If you forget `#1`, the word disappears (until we learn how to use the exclamation mark), as can be seen in **example 2**.

Note that there's still a space between *forgotten* and the full stop. Where does it come from? Well, it is the space that was between *forgotten* and *something*. At the time when XƎSearch manipulates *something*, this space has already been read and typeset, so it does not disappear.

But there's something much more interesting in this example. As you might have noticed, the first line says:

```
\SearchList{list1}{\italics{#1}}{obviously}
```

and in the text to be searched we find '`Obviously`', with an uppercase first letter. Nonetheless, it is found and modified according to the replacement text. We thus discover one basic principle of XƎSearch: *it isn't case-sensitive by default*. Hence the two following lists

```
\SearchList{list1}{<whatever>}{word}
\SearchList{list2}{<whatever>}{Word}
```

will find exactly the same set of words, namely '`word`' '`Word`', '`woRd`', '`WORD`', etc. How scary. This isn't customary in good programming and in TEX in particular. Fortunately, this default setting can be easily changed: the optional star just after `\SearchList` will make the entire list case-sensitive. Besides, if a list is not case-sensitive, i.e. if it has no star, a star before a word in that list will make the search for that particular word case-sensitive.[2] This is illustrated in **example 3**.

In this example we discover another macro, whose meaning is clear:

- **\StopList**{⟨***list of lists***⟩}

The lists, separated by commas, are turned off.

[2]However, if `\SearchList` is suffixed with a star, all words in the list will be case-sensitive.

```
\SearchList{list1}{\italics{#1}}{obviously}
\SearchList{list2}{}{something}
Obviously, I have forgotten something.
```

*Obviously*, I have forgotten .

**Example 2: Words As Arguments**

```
\SearchList{Case insensitive}{\blue{#1}}{woRd}
Word word woRd WORD
\StopList{Case insensitive}
```

Word word woRd WORD

```
\SearchList*{Case sensitive}{\red{#1}}{word}
Word word woRd WORD
\StopList{Case sensitive}
```

Word word woRd WORD

```
\SearchList{Mixed}{\green{#1}}{word,*Worm}
Word word woRd WORD\par
Worm worm woRm WORM\par
```

Word word woRd WORD
Worm worm woRm WORM

**Example 3: Illustrating Case-Sensitivity**

Let's turn back to `\SearchList` again. It can also take an exclamation mark beside the star (the order between the two of them is not important). In this case, the word is not subsituted anymore; i.e. the replacement text will follow the word (still with `#1` standing for it). These concatenating replacements are very dangerous because they are expanded *after* the search has started again. You see what I mean: if the word you've found does not endure some transformation that'll make it different from itself as far as typesetting is concerned, ooops, here's the loop. `WORD` expands to `WORD\command{WORD}` to `WORD\command{WORD\command{WORD}}`, etc., and there's no way out of it.

So, what's the point? The point is: the reason why those replacements are placed after the no-search area has stopped is because they are meant to host argument-taking commands to act on the rest of the streams. Such commands can't be placed in normal replacement texts without an exclamation mark, because they would stumble upon precisely what starts the search again. So be careful. Either use `!`-marked searches with non-typesetting macros, for instance to index the word, or make sure that you know exactly the many interactions you might create. The exclamation mark says it all. **Example 4** is silly but I hope you can see the point.

```
\SearchList*!{Hamlet}%
            { Or Not \StopSearching#1\StartSearching}%
            {To Be}
To Be...
```

To Be Or Not To Be...

```
\SearchList!{typo}{\red{!!!}}{tipo}
There's a tipo here.
```

There's a tipo!!! here.

```
\SearchList!{XeTeX}{ \green}{is}
This is \XeTeX.\par
```

This is XƎTEX.

**Example 4: A Silly One**

Note the space at the beginning of the first and third replacement texts. Concatenating replacement texts (which replace nothing but whatever) stick to their targets. Besides, in the third example, `\green` would have gobbled the subsequent space.

I hope you have noticed that the `Hamlet` list contains not a word but a phrase. So you know: XƎSearch can find phrases. Now we can't avoid going into a little more detail concerning the way XƎSearch works. But before that, let's see one simple macro:

- **\AddToList⟨*!⟩{⟨name⟩}{⟨list of words⟩}**

This adds the words to the ⟨*name*⟩ list, which of course should already exist. The presence or absence of a star and/or an exclamation mark doesn't depend at all on the original list. You can see that in **example 5**.

```
\SearchList{Stupid list}{\blue{#1}}{word}
Word and beep.
\AddToList*{Stupid list}{Beep}
Or Beep and word and beep.
```

Word and beep. Or Beep and word and beep.

**Example 5: Adding Words To An Existing List (Another Silly One)**

Finally, the words in `\SearchList` and `\AddToList` should be made of characters only, but these can be the product of expansion. For instance, if you have `\def\word{a word}`, then you can say `\AddToList{mylist}{\word}`. If anything else shows up XƎSearch won't accept the word (and you'll probably have a good deal of errors beforehand).

## 3 What XƎSearch looks for and how it finds it

XƎSearch can see only two things: letters and non-letters. Non-letters it doesn't like because it's then forced to spit the letters it has gathered and form a word, and most times it's not allowed to take it away. (Un)fortunately, XƎSearch is quite short-sighted: it considers letters what you tell it are not non-letters (XƎSearch apparently has some formal education in philosophy).

More seriously (and clearly), XƎSearch forms a word as long as there are letters. As you can see in **example 6**, macros are expanded and if they yield letters, XƎSearch can recognize a word. So when does it stop searching? There are two main cases:
1. It encounters a space, or any primitive control sequence. The former case is quite natural: you want spaces to delimit words (including `\skip`s and associates). But the latter is less obvious: as soon as TEX does something that is not typesetting letters, XƎSearch gives up. And this includes something as seemingly innocuous as a `\relax`, as you can see in **example 7**. That's the reason why, for instance, XƎSearch will never find `TeX` in `\TeX`: the definition contains many operations that aren't strictly speaking putting letters in the stream. Fortunately, the bulk of a manuscript is made of letters and spaces, and one seldom inserts `\relax`es in the middle of words.
2. XƎSearch encounters a character that you've declared as a non-letter, that is a word boundary. This leads us to the following macro:

- **\MakeBoundary**{⟨***characters***⟩}
- **\UndoBoundary**{⟨***characters***⟩}

The characters should be simply put one after the other, as in for instance

```
\MakeBoundary{,;:!}
\UndoBoundary{?()\{\}}
```

The basic set of such characters is as follows[3]

```
.,;:!?-`'()[]{}
```

Now, if XƎSearch encounters a character that you've made into a boundary, it will stop forming a word and evaluate what it has gathered. Conversely, such characters cannot appear in the list of words in `\SearchList`; they wouldn't be found anyway. This is illustrated in **example 8**.

There is one big difference between those two cases. Characters defined as boundaries are not only word boundaries but also phrase boundaries. If XƎSearch smells a possible phrase, spaces and primitive commands won't stop it, whereas boundary characters will.

[3]That is: full stop, comma, semi-colon, colon, exclamation mark, question mark, dash, inverted comma, apostrophe (i.e. left and right quote), parentheses, brackets, curly braces. This is rather arbitrary, despite some basic sensible assumptions.

```
\SearchList{Will it find me?}{\blue{#1}}{word}
\def\rd{rd}
Here is a wo\rd.
```

Here is a word.

**EXAMPLE 6: MACROS CAN'T HIDE LETTERS**

```
\SearchList{This time I'm prepared}{\blue{#1}}{word}
\def\rd{\relax rd}
Here is a wo\rd.
```

Here is a word.

**EXAMPLE 7: BUT PRIMITIVE CAN**

```
\MakeBoundary{/}
\SearchList{separated}{\ddag#1\ddag}{waka,jawaka}
Waka/Jawaka
```

‡Waka‡/‡Jawaka‡

```
\UndoBoundary{/}
\SearchList{united}{\ddag#1\ddag}{waka/jawaka}
Waka/Jawaka
```

‡Waka/Jawaka‡

**EXAMPLE 8: WHERE WORDS START AND STOP**

You can see that in **example 9**. This example also illustrates one fact and one sad truth. The fact is that words aren't searched for inside phrases; so the first two *you*'s were not turned to italics, since they belonged to *you are what you is*. The third one, one the other hand, was recognized since *you are neither good nor bad* was missed because of the intervenig comma.

The sad truth is that the `\kern` disappeared. This is one shortcoming of XƎSearch: primitives disappear when they're in the middle of a possible phrase, even if that phrase is not recognized in the end. By 'possible phrase' I mean a string of words that form the beginning of a phrase that you want identified, e.g. the kern in

```
\SearchList{H(a)unting primitives}{<whatever>}%
           {xesearch feeds on kerns}
xesearch feeds on\kern1cm skips
```

will disappear, even though no string matches in the end. Hopefully such commands are rather rare in the bulk of a document. If some are unavoidable — and for other uses too — there exists a pair of commands, whose function I probably don't need to explain (except that `\StartSearching` doesn't need to be issued at the beginning of your document, it is there by default):

- **\StartSearching**
- **\StopSearching**

```
\SearchList{word}{\italics{#1}}{you}
\SearchList{phrases}{\red{#1}}
                    {you are what you is,
                     you are neither good nor bad}

You are what\kern1cm % What a kern!
you is but you are neither good, nor bad.
```

You are what you is but *you* are neither good, nor bad.

**Example 9: Phrases And Words**

## 4 (A very blunt form of) regular expressions

Words are cool, and phrases too. But life doesn't always reach their level of achievement. Sometimes you don't know what you want. Something beginning with a 'B', why not? or maybe something that ends in 'et'? Then look at **example 10**.

There are several things to see in this example. First, XƎSearch has entered the `\italics` command and imposed its will.[4] Next, affixes[5] are also sensitive to case-sensitivity, so to speak, since *beside* was not identified (`*B?` being case-sensitive), whereas *PET* was found (`?et` not being case-sensitive). Note that a word matches an affix search if it is at least as

```
\SearchList{Affixes}{\red{#1}}{*B?,?et,?ET}

A \italics{Black Page} in B, actually some kind of
duet for Terry Bozzio and Chad Wackerman, lay
on the drumset beside the PET facility.
```

A *Black Page* in B, actually some kind of duet for Terry Bozzio and Chad Wackerman, lay on the drumset beside the PET facility.

**Example 10: Prefixes And Suffixes**

[4] Provided I'm using commands that don't cancel each other, like plain TEX's `\bf` and `\it`.

[5] I use the word *affixes* to refer to both *prefixes* (like `B?`) and *suffixes* (like `?et`). From a linguistic point of view, prefixes and suffixes (and infixes, actually) are indeed affixes, but from the same point a view, what we're talking about here has nothing to do with prefixes or suffixes, just with bits of words. I hope you don't mind.

long as the specified part of the affix. Thus, *B* matches `B?`. So the question mark means 'from zero to any number of additional letters,' and not 'at least one additional letter.'

Phrases can take only suffixes, and they affect the last word only. So

```
\SearchList{list}{<whatever>}{some interesting wor?}
```

will find `some interesting world`, `some interesting words`, but not `some interesting word thesaurus`. An affix mark anywhere else will have no effect.

Marking the unspecified part of a word with `?` is the only possibility for the question mark to enter a `\SearchList`, and obviously it doesn't stand for itself. So, unless of course you undo it as a string boundary, `?` can appear only at the beginning or the end of a word.[6] In any other place, it will be recognized as a boundary that has no right to be there and you'll be blamed. This means that infixes don't exist in X∃Search, i.e. you can't say `B?et` to search for `bullet`, for instance. Also, you can't say `?ull?` to match `bullet`. One affix at a time.

Finally, don't try to use a joker, i.e.

```
\SearchList{list}{<whatever>}{?}
```

as an attempt to match all words. This won't work.[7]

## 5 Search order(s)

Now we shall see what happens when a word is matched by several searches. There are three different cases:

1. A word is matched by two or more strictly identical searches, e.g.:

```
\SearchList{list1}{<whatever>}{word}
\SearchList{list2}{<whatever else>}{word}
... word ...
```

2. A word is matched by two or more prefixes or two or more suffixes identical in case-sensitivity, e.g.:

```
\SearchList{list1}{<whatever>}{*wor?}
```

[6]And if a star is present, it should precede the question mark.

[7]If you want to match all words
`\SearchList{list}{<whatever>}{a?,b?,...,z?}`
should do. Ok, now you've read it, you might have the impression that the title of this section verges on dishonesty. You might be right.

```
\SearchList{list2}{<whatever else>}{*wo?}
... word ...
```

3. A word is matched by two or more different searches, e.g.:

```
\SearchList{list1}{<whatever>}{*wor?}
\SearchList{list2}{<whatever else>}{word}
\SearchList{list3}{<anything>}{?ord}
... word ...
```

### 5.1 Strictly identical searches

In this case, the word will execute all the replacement texts. Their interactions depend on the way they are defined: the replacement texts that are defined without an exclamation mark take as arguments the replacement texts that are defined just before them and will themselves become arguments to subsequent replacement texts. See **example 11**

If the replacement texts are defined with and exclamation mark, they are simply concatenated, and most importantly, their argument is the word itself alone, not the accumulation of previous remplacement texts. See **example 12**. Of course, if a word is matched by both kinds of replacement texts, the same rules apply, as in **example 13**, where you can also be entertained by some not-very-fun-but-you-can-hopefully-see-the-point-again fiddling with !-marked macros. If you want to know what those three `\expandafter`s are doing here, see section 6.

### 5.2 Affixes with identical characteristics

When a word is found by two or more affixes of the same kind (i.e. only prefixes or only suffixes) and with the same case-sensitivity, then you decide. XƎSearch provides the following commands:

- **\SortByLength**⟨*⟩{⟨*pPsS*⟩}
- **\DoNotSort**{⟨*pPsS*⟩}
- **\SearchAll**{⟨*pPsS*⟩}
- **\SearchOnlyOne**{⟨*pPsS*⟩}

`p`, `P`, `s` and `S` are shorthands for (respectively) 'case-insensitive prefix', 'case-sensitive prefix', 'case-insensitive suffix' and 'case-sensitive suffix'. They refer to the type of affix to modify and those commands can take one or several of them, e.g. `\SearchAll{pSP}`. By

```
\SearchList{list1}{\blue{#1}}{blue word}
\SearchList{list2}{\dag#1\dag}{blue word}
\SearchList{list3}{\ddag#1\ddag}{blue word}

This blue word wears earrings and is equivalent
to \ddag\dag\blue{term}\dag\ddag.
```

This ‡†blue word†‡ wears earrings and is equivalent to ‡†term†‡.

**Example 11: Nested Replacement Texts**

```
\SearchList!{list1}{+}{wor?}
\SearchList!{list2}{\dag}{wor?}
\SearchList!{list3}{\ddag}{wor?}
This word is a freight train.
```

This word+†‡ is a freight train.

**Example 12: Concatenation (Yet Another Silly Example)**

```
\SearchList{list1}{\green{#1}}{*?ORD}
\SearchList{list2}{\ddag#1\ddag}{*?ORD}
\def\whisper#1{\italics{ (#1)}}
\def\ingreen{in green}
\SearchList!{list3}
            {\expandafter\expandafter\expandafter\whisper}
            {*?ORD}
\SearchList!{list4}{\ingreen}{*?ORD}
This WORD must be upset.
```

This ‡WORD‡ *(in green)* must be upset.

**Example 13: Everything Together (This Is Mind-Blowing)**

default, affixes follow the same rules as full words: each replacement text will take the replacement text defined just before as argument. But you can also create an order between them: with `\SortByLength`, longer affixes match words before shorter ones, and their replacement texts are thus more deeply nested; adding a star to `\SortByLength` reverses the order: shorter affixes before longer ones. `\DoNotSort` resets to default, i.e. replacement texts follow the order in which they were defined. See **example 14**.

`\SearchAll` and `\SearchOnlyOne` sets what should happen when a word is matched by an affix: shall the search stop, or shall X∃Search continue to investigate whether other affixes might fit too? By default, all affixes are tested, but you might want a different behavior. Thus `\SearchOnlyOne{PS}` will make case-sensitive prefixes and suffixes search only once (and thus the order defined just before becomes extremely important) while `\SearchAll{PS}` will return to default, as illustrated in **example 15**.

### 5.3 Different searches

Finally, we have to see what X∃Search should do when several searches match a word. Once again, you decide, thanks to the following command:

- **\SearchOrder**{⟨***order and inhibitions***⟩}

You know what `p`, `P`, `s` and `S` mean; `f` and `F` mean 'case-insensitive full word' and 'case-sensitive full word.' In the macro above, ⟨*order and inhibitions*⟩ is a list of one or more sequences like `f!ps;` (with the semi-colon as part of the expression) in which the red part is optional and which means: if a word matches a full-word case-insensitive search, then X∃Search will not test case-insensitive prefixes and suffixes on this word. Such declarations are put one after the other, and this defines the search order. For instance, the default order for X∃Search is:

```
\SearchOrder{
  F!fPpSs;
  f!PpSs;
  P!pSs;
  p!Ss;
  S!s;
  s;
  }
```

and it simply means that full words should be searched for before prefixes, and prefixes before suffixes, with case-sensitive search first in each case, and that any successful search

```
\SearchList{Three letters}{\ddag#1\ddag}{*adv?}
\SearchList{Two letters}{\red{#1}}{*ad?}
\SearchList{Four letters}{\dag#1\dag}{*adve?}

\SortByLength{P} adverb
\SortByLength*{P} adverb
\DoNotSort{P} adverb
```

‡†adverb†‡ †‡adverb‡† †‡adverb‡†

**Example 14: This Is Fascinating**

```
\SearchList{just a list}{\blue{#1}}{bl?,*bo?}
\SearchList{just another list}{\bold{#1}}{blu?,*bol?}

\SearchOnlyOne{P} Blue and bold and
\SortByLength{P} bold and blue.
```

**Blue** and bold and **bold** and **blue**.

**Example 15: This Guy Sure Ain't No David Foster Wallace**

inhibits any subsequent test. You can have as many sequences as you wish. If XƎTEX goes crazy and never terminates, then you've probably forgotten a semi-colon (I do it very frequently). See **example 16** for an illustration.

Remember that e.g. `word?` will find '`word`' as a prefix, not as a full word, so that '`word`' will not be found if you say for instance `\SearchList{list}{<whatever>}{word?}` and `\SearchOrdef{f;}`. Finally, although something like `\SearchOrder{f;}` is perfectly okay to search for case-insensitive full words only, `\SearchOrder{;}` will only make XƎTEX crazy; `\StopSearching` is simpler.

```
\SearchList{word}{\green{#1}}{*Word}
\SearchList{prefix}{\frame{#1}}{wor?}
\SearchList{suffix}{\reverse{#1}}{?ord}

\SearchOrder{F;p;s;}
This Word is well-matched.

\SearchOrder{F!p;p;S;}
This Word is not so well-matched anymore.

\SearchOrder{f;}
This Word is not matched at all.
```

This $\boxed{\text{broW}}$ is well-matched.
This Word is not so well-matched anymore.
This Word is not matched at all.

**Example 16: Search Order**

## 6 Some TEXnical matters

This section is not vital to the comprehension of XƎSearch, but it may be useful.

- **\PrefixFound**
- **\SuffixFound**
- **\AffixFound**

When a word is found thanks to an affix search, the prefix or suffix used is stored in the relevant macros. If there are several matching affixes, the last prefix and the last suffix win in their respective categories, and between them the same rule apply for `\AffixFound`. These macros are available as long as the search has not started again, i.e. they're fully available in normal replacement texts, but in `!`-marked definitions they're erased as soon as a letter is typeset, so they can be used only at the very beginning. The rest of the time they are empty.

The affix itself respects the case in which it was declared if it is case-sensitive, but it is in lowercase otherwise, however it was fed to `\SearchList`. See **example 17**.

```
\SearchList{A case-sensitive suffix}{Suf\blue\SuffixFound}{*?FiX}
SufFiX.
```

SufFiX.

```
\SearchList{A case-insensitive affix}{\blue\AffixFound fix}{Pre?}
PREfix.
```

prefix.

**Example 17: Finding Affixes**

- **\PatchOutput**
- **\NormalOutput**

By default, XƎSearch doesn't patch the output routine so footers and headers are searched. This can be done by these two commands. `\PatchOutput` should of course be issued after any modification to the output routine. `\NormalOutput` restores the value of the output routine at work when `\PatchOutput` was executed.

- **\PatchTracing**
- **\NormalTracing**

If you want to give a look at your log file with some tracing on, you will find hundreds if not thousands of totally uninformative lines. That's XƎSearch recursively discovering new letters and testing words. With `\PatchTracing`, XƎSearch will try to keep quiet during those painful moments, i.e. `\tracingcommands` and `\tracingmacros` will be turned to zero. It can't exactly be totally silent, so just know that all its words begin with `xs@`. `\NormalTracing` lets XƎSearch express itself again.

Now just consider **example 18**. When XƎSearch reads the input, it introduces itself to all the letters it doesn't know. Most importantly, it writes down some information about them, like their catcode. Now, if a letter is met with a given category catcode, that's the way XƎSearch will remember it, and this will influence how prefixes and suffixes are recognized. More precisely: the identification of a letter (e.g. the first occurence of it in the typestting stream) and its definition as part of an affix should be done under the same category code.

Note that in **example 18** I first had to stop the `fz` list, otherwise the prefix `Frank Zap?` would not have been recreated. Another solution would have been to create another prefix like `Frank Za?` or `*Frank Zap?`.

Finally, here's how replacement texts are processed. Suppose you have:

```
\SearchList{listone}{\italics{#1}}{word}
\SearchList{listtwo}{\blue{#1}}{word}
\SearchList{listthree}{\bold{#1}}{word}
```

then XƎSearch does something like this:

```
\def\command@listone#1{\italics{#1}}
\def\command@listtwo#1{\blue{#1}}
\def\command@listthree#1{\bold{#1}}
```

and when `word` is encountered it is turned to

```
\expandafter\command@listthree\expandafter{%
  \expandafter\command@listtwo\expandafter{%
    \expandafter\command@listone\expandafter{\WORD}}}
```

where `\WORD` contains exactly `word`; as you can see, this is equivalent to

```
\command@listthree{\command@listtwo{\command@listone{word}}}
```

which you won't have failed to notice is not equivalent to

```
\bold{\blue{\italics{word}}}
```

```
\catcode`\Z=12
Here's a Z.
\catcode`\Z=11

\SearchList{fz}{\italics{#1}}{Frank Zap?}
Look, here comes Frank Zappa!

\StopList{fz}
\catcode`\Z=12
\SearchList{true fz}{\italics{#1}}{Frank Zap?}
One more time for the world.
Here comes Frank Zappa!
```

Here's a Z.
Look, here comes Frank Zappa!
One more time for the world. Here comes *Frank Zappa*!

**Example 18: The Mysterious Z**

although in this example the difference is immaterial. Now, if you really want three expansions with superior precision on one word, you probably don't need XƎSearch: just use a good old macro instead.

Finally, !-marked replacement texts are simply concatenated, as in:

```
\expandafter\command@listone\expandafter{\WORD}
\expandafter\command@listthree\expandafter{\WORD}
\expandafter\command@listtwo\expandafter{\WORD}
```

Now you can see the reason for the three `\expandafter`'s in **example 13**.

## 7 Examples

XƎSearch was first designed as the basis for the XƎIndex package, an automatic indexing package for XƎLaTeX. It developped into a stand-alone project, and standing so alone that there are no other application yet. So here are some ideas.

First, this document has the following list:

```
\SearchList*{logos}{\csname#1\endcsname}{?TeX,?ConTeXt,xesearch}
```

(with `\xesearch` properly defined beforehand) so throughout this document I was able to type '`xesearch can do this or that`' to produce 'XƎSearch can do this or that'. That's not fascinating but it was a test.

Being a linguist I can also directly paste examples from my database and turn on XƎSearch to highlight some words. For instance, suppose you're studying the grammaticalization of, say, *going to* in English,[8] and you have many examples. Then you just create a command like `\startexample`, or patch an existing command to activate XƎSearch just for this stretch of text, among other things. For instance:

```
\SearchList{goingto}{\bold{#1}}{going to}
\def\startexample{%
  Here you can modify margins, for instance.
  \StartSearching
  }
\def\stopexample{%
  \StopSearching
```

[8] If you're a linguist, I apologize for my lack of originality.

```
Here you restore previous values.
}
```

Otherwise you can locally use \StopList if you're searching the rest of the document too.

What follows are some sketchy ideas. Concerning syntax highlighting, I won't try to compete with the listings package.

## 7.1 Spelling

Here's a recipe to create an English spellchecker. Take the list of the 40,000 most frequent words of English by Wiktionary: http://en.wiktionary.org/wiki/Wiktionary:Frequency_lists#English. Use TEX to turn it into a file, say english.dic, whose only content is \csname*<word>*@dic\endcsname for each word of the list, with *<word>* in lowercase. What! you exclaim, that creates 40,000 control sequences! True. But TEX distributions can easily do that today. Input english.dic at the beginning of your document. Then set up X∃Search as follows:

```
\SearchList{spelling}{%
  \lowercase{\ifcsname#1@dic\endcsname}%
    #1%
  \else
    \red{#1}%
  \fi}
  {a?,b?,c?,d?,e?,f?,g?,h?,i?,j?,k?,l?,m?,
  n?,o?,p?,q?,r?,s?,t?,u?,v?,w?,x?,y?,z?}
\SearchOrder{p;}
```

Now, for each word, X∃Search checks whether it belongs to the frequency list. If it doesn't, it puts it in red, thus signaling a likely spelling error. It could also issue an error message, or whatever.

Some words will never belong to that list. Then we use a simple macro to add them beforehand:

```
\def\AddWord#1{\lowercase{\csname#1@dic\endcsname}}
```

We could also create more specific macros like \AddRegularVerb which from e.g. change would add change, changes, changed, changing. TEX could also rewrite english.dic on the fly so there'd be no need to respecify those words on every document. And so on and so forth.

Stately, plump Buck Mulligan came from the stairhead, bearing a bowl of lather on which a mirror and a razor lay crossed. A yellow dressinggown, ungirdled, was sustained gently behind him on the mild morning air. He held the bowl aloft and intoned:
— *Introibo ad altare Dei*.
Halted, he peered down the dark winding stairs and called out coarsely:
— Come up, Kinch! Come up, you fearful jesuit!
Solemnly he came forward and mounted the round gunrest. He faced about and blessed gravely thrice the tower, the surrounding land and the awaking mountains. Then, catching sight of Stephen Dedalus, he bent towards him and made rapid crosses in the air, gurgling in his throat and shaking his head. Stephen Dedalus, displeased and sleepy, leaned his arms on the top of the staircase and looked coldly at the shaking gurgling face that blessed him, equine in its length, and at the light untonsured hair, grained and hued like pale oak.
Buck Mulligan peeped an instant under the mirror and then covered the bowl smartly.
— Back to barracks! he said sternly.
He added in a preacher's tone:
— For this, O dearly beloved, is the genuine Christine: body and soul and blood and ouns. Slow music, please. Shut your eyes, gents. One moment. A little trouble about those white corpuscles. Silence, all.

**Example 19: The Words In Red Don't Belong To The Top 40,000**

Using a list like the frequency list is important because we want all forms of a word to appear; i.e. organized word lists have `hear` and not `hears`, because there exists either an algorithm or at least the user's brain to derive `hears` from `hear`.

## 7.2 Word count

Another simple use of X∃Search is counting words in a document. We define a case-insensitive list with all letters as prefixes, so all words will be matched (we could add numbers too), as we did in the previous example. Supposing we want words like *don't* to be counted as one word, then we remove the apostrophe from the word boundaries (in case it signals a dialogue, the following space will delimit the word anyway). And we define the search order as case-sensitive prefixes only, because we don't need anything else. The `\shownumber` macro is clear, I believe. In the first version of the text on the right it is `\let` to `\relax`. It's just for fun.

The `\advance` on `\wordcount` has to be `\global` because there might be (hidden) groups in the text, for instance in font-changing commands.

```
\newcount\wordcount
\def\shownumber{%
  \raise.6\baselineskip\hbox to0pt{\hss\tiny\red{\the\wordcount}}
  }
\SearchList!{wordcount}{\global\advance\wordcount1\shownumber{}}
  {a?,b?,c?,d?,e?,f?,g?,h?,i?,j?,k?,l?,m?,
  n?,o?,p?,q?,r?,s?,t?,u?,v?,w?,x?,y?,z?}
\UndoBoundary{'}
\SearchOrder{p;}
```

Stately, plump Buck Mulligan came from the stairhead, bearing a bowl of lather on which a mirror and a razor lay crossed. A yellow dressinggown, ungirdled, was sustained gently behind him on the mild morning air. He held the bowl aloft and intoned:
— *Introibo ad altare Dei.*
Halted, he peered down the dark winding stairs and called out coarsely:
— Come up, Kinch! Come up, you fearful jesuit!
Solemnly he came forward and mounted the round gunrest. He faced about and blessed gravely thrice the tower, the surrounding land and the awaking mountains. Then, catching sight of Stephen Dedalus, he bent towards him and made rapid crosses in the air, gurgling in his throat and shaking his head. Stephen Dedalus, displeased and sleepy, leaned his arms on the top of the staircase and looked coldly at the shaking gurgling face that blessed him, equine in its length, and at the light untonsured hair, grained and hued like pale oak.

*There are 158 words.*

Buck[159] Mulligan[160] peeped[161] an[162] instant[163] under[164] the[165] mirror[166] and[167] then[168] covered[169] the[170] bowl[171] smartly[172].
— Back[173] to[174] barracks[175]! he[176] said[177] sternly[178].
He[179] added[180] in[181] a[182] preacher's[183] tone[184]:
— For[185] this[186], O[187] dearly[188] beloved[189], is[190] the[191] genuine[192] Christine[193]: body[194] and[195] soul[196] and[197] blood[198] and[199] ouns[200]. Slow[201] music[202], please[203]. Shut[204] your[205] eyes[206], gents[207]. One[208] moment[209]. A[210] little[211] trouble[212] about[213] those[214] white[215] corpuscles[216]. Silence[217], all[218].

*The total number of words is: 218.*

**Example 20: Counting Words**

## 7.3 Syntax highlighting: TEX

At first I'd designed a colorful scheme but it was ugly, so here's something much more sober. We simply create an empty list in which we design a macro to add `\stringed` primitive commands.

```
\SearchList{hilitex}{\bold{#1}}{}
\def\Add#1{%
  \AddToList{hilitex}{#1}%
  }
```

```
\expandafter\Add\expandafter{\string\def}
\expandafter\Add\expandafter{\string\expandafter}
\expandafter\Add\expandafter{\string\else}
\expandafter\Add\expandafter{\string\fi}
\expandafter\Add\expandafter{\string\else}
```

We can't do that for prefixes (and we need them if we want e.g. to underline all user-defined `\if`), because they would be `\stringed` and thus of category code 12, which **example 18** has shown was a trouble. So we design a macro to add words with a backslash added beforehand. And we use it.

```
\def\gobble#1{}
\def\AddPrefix#1{%
  \AddToList*{hilitex}{\expandafter\gobble\string\\#1?}%
  }
\AddPrefix{new} \AddPrefix{if}
```

We need one last thing. We want \ to be recognized as a letter, because it should be put in bold too. But we also want it to be recognized as a string boundary. The only solution is to make it active and let it expand to `\relax` (a natural string boundary) plus itself in catcode 12 (which is not defined with `\MakeBoundary` and is thus a letter for XƎSearch).

```
\catcode`\|=0
\catcode`\\=13
|def\{|relax|string\}
```

If we pack everything into an usual macro to make verbatim text, then we obtain something along the lines of **example 21**. Don't forget the typewriter font for the real thrill!

The implementation section of this documentation displays a subtler kind of syntax highlighting, viz. `\def` and associates put the following command in red and index it too, except commands I don't want to see treated as such, like temporary commands. However, the implementation depends on CodeDoc's macros, so I won't show it here, although you can look at the source.

```
\def\mycommand#1{%
  \expandafter\myothercommand#1%
  \ifwhatever
    \newtoks\mytoks
    \mytoks={...}%
  \else
    \mytoks={...}%
  \fi
  }
```

**Example 21: TEX Highlighted**

## 7.4 Syntax highlighting: HTML

Coloring HTML is rather easy. The most complicated part concerns word boundaries. XƎSearch is used to find elements and attributes. Only case-insensitive full words need to be searched for.

```
\MakeBoundary{<>/=}
\SearchList{elements}{\bold{\violet{#1}}}
           {html,meta,head,body,span,p,div,b,h1,img}
\SearchList{attributes}{\bold{#1}}{align,class,style,src}
\SearchOrder{f;}
```

< and > delimit markup, so we use them to switch XƎSearch on and off.

```
\catcode`\<=13
\catcode`\>=13
\def<{\bgroup\catcode`\'=13\catcode`\"=13\char`\<\StartSearching{}}
\def>{\egroup\char`\>}
```

Quoted text should not be searched, because values to attributes are simply put in blue. Double quotes and single quotes should exclude each other.

```
\catcode`\"=13
\newif\ifdbbegin
\def"{%
  \unless\ifsgbegin
    \ifdbbegin \egroup \char`\"
    \else \char`\" \bgroup \dbbegintrue \color{blue}\StopSearching
    \fi
  \fi
  }
\catcode`\'=13
\newif\ifsgbegin
\def'{%
  \unless\ifdbbegin
    \ifsgbegin \egroup \char`\'
    \else \char`\' \bgroup \sgbegintrue \color{blue}\StopSearching
    \fi
  \fi
  }
```

`src` and `href` take links as values, usually underlined. So we do just that.

```
\SearchList!{links}{\makelink}{src,href}
\def\makelink=#1{%
```

```
  \ifx#1"
    \expandafter\makedbqlink
  \else
    \expandafter\makesgqlink
  \fi
  }
\def\makedbqlink#1"{\StopSearching="\underline{#1}"\StartSearching}
\def\makesgqlink#1'{\StopSearching='\underline{#1}'\StartSearching}
```

The `&...;` character denotation is often in red.

```
\catcode`\&=13
\def&#1;{%
  \char`\&
  \red{#1;}%
  }
```

Finally we turn off TEX's special characters (quotes are made active by < and >), and we make some useful adjustments.

```
\catcode`\"=12
\catcode`\'=12
\catcode`\#=12
\catcode`\_=12
\catcode`\^=12
\catcode`\%=12
\obeylines
\def\par{\leavevmode\endgraf}
\parindent0pt
```

**Example 22** shows the bottom of the CTAN page.

```
...
<p>
A perhaps less taxing way to express your appreciation
is to make a
<a href ="https://www.tug.org/donate.html#ctan">donation</a> &mdash;
small efforts add up </p>
<div id='footer'><hr />
<table width='100%'>
  <tr>
    <td align='left'>
      <span id='footer_author'>Site sponsor:
     <a href='http://www.tug.org'>TeX Users Group</a></span></td>
    <td>
      <span id='footer_middle'>Internet connection provided by
     <a href='http://www.smcvt.edu'>St Michael's College</a></span></td>
    <td align='right'>
      <span id='footer_home'>
     <a href='/what_is_ctan.html'>What is CTAN?</a></span></td>
  </tr>
</table>
</div>
</body>
</html>
```

**Example 22: Colorful HTML**

# 8 Implementation

## 8.1 First things first

First we look for XƎTEX.

```
\ifx\csname XeTeXrevision\endcsname\relax
  \errmessage{You need XeTeX to run xesearch. It won't be loaded.}
  \expandafter\endinput
\else
  \expandafter\ifx\csname xs@ChangeCatcodes\endcsname\relax
  \else
    \expandafter\expandafter\expandafter\endinput
  \fi
\fi
```

These will be used to keep a constant punctuation in spite of catcode-changing packages like `babel`.

```
\catcode`@=11
\def\xs@ChangeCatcodes{%
  \chardef\xs@questioncode=\catcode`\?%
  \chardef\xs@exclamationcode=\catcode`\!%
  \chardef\xs@commacode=\catcode`\,%
  \chardef\xs@starcode=\catcode`\*%
  \chardef\xs@semicoloncode=\catcode`\;%
  \catcode`\?12
  \catcode`\!12
  \catcode`\,12
  \catcode`\*12
  \catcode`\;12
  }
\def\xs@RestoreCatcodes{%
  \catcode`\?\xs@questioncode
  \catcode`\!\xs@exclamationcode
  \catcode`\,\xs@commacode
  \catcode`\*\xs@starcode
  \catcode`\;\xs@semicoloncode
  }
\xs@ChangeCatcodes
```

We declare XƎSearch as a package in LATEX.

```
\ifdefined\ProvidesPackage
  \def\xs@err#1{\PackageError{xesearch}{#1}{}}
```

```
  \ProvidesPackage{!FileName}[!FileDate!space !FileVersion!space Searching documents.]
\else
  \def\MessageBreak{^^J}
  \def\xs@err#1{%
    \bgroup
    \newlinechar`\^^J%
    \errorcontextlines=0
    \errmessage{xsearch error: #1}%
    \egroup
    }
\fi
```

`\unexpanded` already exists in ConTEXt, and the meaning of the ε-TEX primitive is taken over by `\normalunexpanded`, so we have to make the proper adjustment (many thanks to Wolfgang Schuster, who signalled this to me).
`\xs@contextmodule` is an empty command let to `\relax` when XƎSearch is loaded with ConTEXt.

```
\ifcsname xs@contextmodule\endcsname
  \let\xs@unexpanded\normalunexpanded
\else
  \let\xs@unexpanded\unexpanded
\fi
```

Some keywords, indispensable macros, and a bunch of `\new` things.

```
\def\xs@end{\xs@end}
\def\xs@empty{}
\def\xs@star{*}
\def\xs@exclamation{!}
\def\xs@question{?}
\def\xs@starexclam{*!}
\def\xs@exclamstar{!*}
\def\xs@words{words}
\def\xs@prefixes{prefixes}
\def\xs@suffixes{suffixes}
\def\xs@gobble#1{}
\def\xs@Lowercase#1#2{\lowercase{\def#2{#1}}}
\let\xs@relax\relax
\newcount\xs@TempCount
\newcount\xs@CaseSensitive
\newcount\xs@TempLength
\newcount\xs@Length
\newbox\xs@Box
```

```
\newif\ifxs@Concatenate
\newif\ifxs@String
\newif\ifxs@Affix
\newif\ifxs@Prefix
\newif\ifxs@Suffix
\newif\ifxs@BadWord
\newif\ifxs@Star
\newif\ifxs@Phrase
\newif\ifxs@Match
\newtoks\xs@DefToks
\newtoks\xs@NoReplaceToks
```

## 8.2 Character classes

Basic classes: natural delimiters (spaces and primitives), left and right delimiters (set by `\MakeBoundary`) and the normal class, out of which letters and delimiters will be taken.

```
\chardef\xs@NatDel=255
\chardef\xs@lrDel=254
\chardef\xs@Classes=253
\chardef\xs@Classless=0
\XeTeXinterchartoks\xs@lrDel\xs@Classless={\xs@LearnLetter}
\XeTeXinterchartoks\xs@NatDel\xs@Classless={\xs@LearnLetter}
\XeTeXinterchartoks\xs@NatDel\xs@lrDel{\xs@EndString}
\xs@TempCount\xs@Classes
```

This is how we make boundaries. Note that if the character has a character class of 8 or 9, we don't change it. The interchartoks will be modified, however.

```
\def\xs@Delimiters{}
\def\xs@MakeDel#1{%
  \ifx#1\xs@end
    \let\xs@next\relax
  \else
    \let\xs@next\xs@MakeDel
    \unless\ifnum\the\XeTeXcharclass`#1=7
      \unless\ifnum\the\XeTeXcharclass`#1=8
        \XeTeXcharclass`#1=\xs@lrDel
        \expandafter\def\expandafter\xs@Delimiters\expandafter{\xs@Delimiters#1}%
      \fi
    \fi
  \fi\xs@next}
\xs@MakeDel\{\}.,;:!?[()]-'`\xs@end
```

```
\def\MakeBoundary#1{%
  \xs@MakeDel#1\xs@end
  }
\def\UndoBoundary#1{%
  \xs@UndoBoundary#1\xs@end
  }
\def\xs@UndoBoundary#1{%
  \def\xs@temp{#1}%
  \ifx\xs@temp\xs@end
    \let\xs@next\relax
  \else
    \ifnum\the\XeTeXcharclass`#1=\xs@lrDel
      \def\xs@RemoveFromDelimiters##1#1##2\xs@end{%
        \def\xs@Delimiters{##1##2}%
      }%
      \expandafter\xs@RemoveFromDelimiters\xs@Delimiters\xs@end
    \fi
    \XeTeXcharclass`#1=0
    \let\xs@next\xs@UndoBoundary
  \fi\xs@next
  }
```

This is the macro that turn a letter into a letter recording itself. It is recursive. Each new letter is assigned a new character class (from 253 downward), then it is made to start the recording process after delimiters, to stop it before, and to add itself to `\xs@String` in both case or next to another letter. Before natural delimiters, however, if the word recorded up to now is part of a possible phrase, the process is not stopped. The `polyglossia` patch is needed when e.g. `?` is not turned into a `\xs@lrDel` but keeps its character class as defined by `polyglossia`.

```
\def\xs@Letters{}%
\def\xs@CreateLetter#1{%
  \ifx#1\xs@end
    \let\xs@next\relax
  \else
    \expandafter\def\expandafter\xs@Letters\expandafter{\xs@Letters#1}%
    \XeTeXcharclass`#1=\xs@TempCount
    \expandafter\def\csname\the\xs@TempCount @xstring@letter\endcsname{#1}%
    \edef\xs@PolyglossiaPatch{%
      \xs@unexpanded{\XeTeXinterchartoks\xs@TempCount7}{%
        \xs@unexpanded{\xdef\xs@String{\xs@String#1}\xs@EndString}%
        \the\XeTeXinterchartoks0 7}%
      \xs@unexpanded{\XeTeXinterchartoks\xs@TempCount8}{%
        \xs@unexpanded{\xdef\xs@String{\xs@String#1}\xs@EndString}%
```

```
    \the\XeTeXinterchartoks0 8}%
  \xs@unexpanded{\XeTeXinterchartoks8\xs@TempCount}{%
    \the\XeTeXinterchartoks8 0 \xs@unexpanded{\xs@StartSring}}%
  }%
\xs@PolyglossiaPatch
\XeTeXinterchartoks\xs@TempCount\xs@Classless{%
  \xdef\xs@String{\xs@String#1}%
  \xs@LearnLetter}%
\XeTeXinterchartoks\xs@lrDel\xs@TempCount{%
  \xs@StopTracing
  \xs@StartString
  }%
\XeTeXinterchartoks\xs@NatDel\xs@TempCount{%
  \xs@StopTracing
  \xs@StartString
  }%
\XeTeXinterchartoks\xs@TempCount\xs@lrDel{%
  \xdef\xs@String{\xs@String#1}\xs@EndString}%
\XeTeXinterchartoks\xs@TempCount\xs@NatDel{%
  \xdef\xs@String{\xs@String#1}%
  \ifcsname\xs@String @xs@phrases@cs\endcsname
    \XeTeXinterchartokenstate0
    \xdef\xs@Stack{%
      \xs@String\noexpand\xs@end\xs@unexpanded\expandafter{\xs@Stack}%
      }%
    \edef\xs@String{\xs@unexpanded\expandafter{\xs@String} }%
    \XeTeXinterchartokenstate1
  \else
    \expandafter\xs@Lowercase\expandafter{\xs@String}\xs@lcString
    \ifcsname\xs@lcString @xs@phrases@ncs\endcsname
      \XeTeXinterchartokenstate0
      \xdef\xs@Stack{%
        \xs@String\noexpand\xs@end\xs@unexpanded\expandafter{\xs@Stack}%
        }%
      \edef\xs@String{\xs@unexpanded\expandafter{\xs@String} }%
```

```
          \XeTeXinterchartokenstate1
        \else
          \expandafter\expandafter\expandafter\xs@EndString
        \fi
      \fi
      }%
    \xs@TempCount\xs@Classes
    \xs@MakeInterCharToks#1%
    \advance\xs@TempCount-1
    \let\xs@next\xs@CreateLetter
  \fi\xs@next
  }
```

This is the recursive macro which creates the `\XeTeXinterchartoks` for the new letter and all existing letter.

```
\def\xs@MakeInterCharToks#1{%
  \ifnum\xs@TempCount=\XeTeXcharclass`#1
    \XeTeXinterchartoks\xs@TempCount\xs@TempCount{\xdef\xs@String{\xs@String#1}}%
    \let\xs@next\relax
  \else\let\xs@next\relax
    \expandafter\expandafter\expandafter%
      \xs@Xict\csname\the\xs@TempCount @xstring@letter\endcsname%
      \xs@TempCount{\XeTeXcharclass`#1}%
    \xs@Xict#1{\XeTeXcharclass`#1}\xs@TempCount
    \advance\xs@TempCount-1
    \def\xs@next{\xs@MakeInterCharToks#1}%
  \fi\xs@next}
\def\xs@Xict#1#2#3{%
  \XeTeXinterchartoks#2#3{\xdef\xs@String{\xs@String#1}}%
  }
```

XƎSearch learns a letter when it encounters a character with character class 0. Since `\xs@CreateLetter` is local, and since it is often executed inside the word box (see `\xs@StartString`), we record the letters thus created in `\xs@PendingLetters` and create them for good after the group.

```
\def\xs@PendingLetters{}%
\def\xs@LearnLetter#1{%
  \xs@CreateLetter#1\xs@end
  \ifxs@String
    \xdef\xs@PendingLetters{\xs@PendingLetters#1}%
  \fi
  #1}
```

## 8.3 Search lists

First we define whether there's an `!` or a `*` or both.

```
\def\SearchList{%
  \xs@ChangeCatcodes
  \xs@StarOrExclam\xs@Search
  }
\def\xs@StarOrExclam#1#2#{%
  \def\xs@temp{#2}%
  \ifx\xs@temp\xs@star
    \xs@CaseSensitive2
    \xs@Concatenatefalse
  \else
    \ifx\xs@temp\xs@exclamation
      \xs@CaseSensitive0
      \xs@Concatenatetrue
    \else
      \ifx\xs@temp\xs@starexclam
        \xs@CaseSensitive2
        \xs@Concatenatetrue
      \else
        \ifx\xs@temp\xs@exclamstar
          \xs@CaseSensitive2
          \xs@Concatenatetrue
        \else
          \xs@CaseSensitive0
          \xs@Concatenatefalse
        \fi
      \fi
    \fi
  \fi#1%
  }
```

Then, after a basic check on the name of the list, we record it and defined the macros associated with this list as the second argument; these macros are the normal and !-marked (‘`noreplace`’) versions (both are created because there might be an `\AddToList` of a different type). Finally we launch the

```
\def\xs@Search#1#2#3{%
  \ifcsname#1@xs@searchlist\endcsname
    \xs@err{%
      `#1' already exists.\MessageBreak
      Use \string\AddToList{#1}{<words>} to add words to it%
```

word-maker on the list of words. `\AddToList` is equivalent with some adjustments.

```
    }%
  \else
    \def\xs@ListName{#1}%
    \expandafter\def\csname\xs@ListName @words\endcsname{}%
    \expandafter\def\csname #1@xs@searchlist\endcsname##1{#2}%
    \expandafter\def\csname #1@xs@searchlist@noreplace\endcsname##1{#2}%
    \expandafter\xs@MakeWord#3,\xs@end,%
    \xs@RestoreCatcodes
  \fi
  }
\def\AddToList{%
  \xs@ChangeCatcodes
  \xs@StarOrExclam\xs@AddToList
  }
\def\xs@AddToList#1#2{%
  \ifcsname#1@xs@searchlist\endcsname
    \def\xs@ListName{#1}%
    \expandafter\xs@MakeWord#2,\xs@end,%
    \xs@RestoreCatcodes
  \else
    \xs@err{`#1' is not a list}%
  \fi
  \xs@RestoreCatcodes
  }
```

This takes each word one by one and checks and creates a few things.

```
\def\xs@MakeWord#1,{%
  \def\xs@TempWord{#1}%
  \ifx\xs@TempWord\xs@end
    \let\xs@next\relax
  \else
    \ifcsname\ifnum\xs@CaseSensitive=2*\fi#1@\xs@ListName\endcsname
      \xs@err{You have already specified `\ifnum\xs@CaseSensitive=2*\fi#1'%
        in `\xs@ListName'. \MessageBreak You can't do it twice}%
    \else
      \csname#1@\xs@ListName\endcsname
      \edef\xs@TempWord{#1}%
```

```
    \chardef\xs@ParseState=0
    \xs@BadWordfalse
    \xs@Starfalse
    \xs@Prefixfalse
    \xs@Suffixfalse
```

For instance, we parse the word, to find prefixes or suffixes or forbidden things, like control sequences. Then we suppress prefixes and suffixes.

```
    \xs@ParseWord#1\xs@end
    \unless\ifxs@BadWord
      \ifxs@Star
        \xs@CaseSensitive1
        \expandafter\xs@SuppressPrefix\xs@TempWord\xs@end
      \fi
      \ifxs@Prefix
        \expandafter\xs@SuppressSuffix\xs@TempWord
      \else
        \ifxs@Suffix
          \expandafter\xs@SuppressPrefix\xs@TempWord\xs@end
        \fi
      \fi
```

Depending on case-sensitivity, we put the word in lowercase or not, and we define a keyword to record the case-sensitivity.

```
      \def\xs@Phrase{}%
      \ifcase\xs@CaseSensitive
        \expandafter\xs@Lowercase\expandafter{\xs@TempWord}\xs@TempWord
        \def\xs@cs{ncs}%
        \expandafter\xs@CheckSpaces\xs@TempWord\xs@end
      \or
        \def\xs@cs{cs}%
        \expandafter\xs@CheckSpaces\xs@TempWord\xs@end
        \xs@CaseSensitive0
      \or
        \def\xs@cs{cs}%
        \expandafter\xs@CheckSpaces\xs@TempWord\xs@end
      \fi
```

Finally, we patch the replacement texts associated with this word or affix.

```
      \ifxs@Prefix
        \xs@MakePrefix
        \def\xs@WordType{prefixes}%
        \expandafter\xs@PatchDef\csname\xs@ListName @xs@searchlist\endcsname
```

This is a basic finite state automaton. It starts in state 0. A star brings it in state 1. In both 0 and 1, if it finds a letter or a ? it goes in state 2. From there, only letters and a ? at the very end of the word are allowed. Boundaries make it crash. The distinction between stage 0 and stage 1 is needed just in case the user defines the star as a boundary.

```
        \else
          \ifxs@Suffix
            \xs@MakeSuffix
            \def\xs@WordType{suffixes}%
            \expandafter\xs@PatchDef\csname\xs@ListName @xs@searchlist\endcsname
          \else
            \def\xs@WordType{words}%
            \expandafter\xs@PatchDef\csname\xs@ListName @xs@searchlist\endcsname
          \fi
        \fi
      \fi
    \fi
    \let\xs@next\xs@MakeWord
  \fi\xs@next
  }
\def\xs@ParseWord#1{%
  \def\xs@temp{#1}%
  \ifx\xs@temp\xs@end
    \let\xs@next\relax
    \ifxs@Suffix
      \ifnum\xs@ParseState=3
        \xs@err{You can't have a prefix and a suffix in the same word.\MessageBreak
          `\xs@unexpanded\expandafter{\xs@TempWord}' won't be searched}%
        \xs@BadWordtrue
      \fi
    \fi
  \else
    \let\xs@next\xs@ParseWord
    \expandafter\ifcat\noexpand#1\relax
      \xs@BadChar#1{control sequences are forbidden}%
    \else
      \ifcase\xs@ParseState
        \chardef\xs@TempNum=\XeTeXcharclass`#1 %
        \ifx\xs@temp\xs@star
          \xs@Startrue
```

```
     \chardef\xs@ParseState=1
     \let\xs@next\xs@ParseWord
    \else
     \ifx\xs@temp\xs@question
      \xs@Suffixtrue
      \chardef\xs@ParseState=2
      \let\xs@next\xs@ParseWord
     \else
      \ifnum\xs@TempNum>\xs@Classes
       \xs@BadChar#1{it's already a string delimiter}%
      \else
       \chardef\xs@ParseState=2
       \ifnum\xs@TempNum=0
        \xs@CreateLetter#1\xs@end
        \let\xs@next\xs@ParseWord
       \fi
      \fi
     \fi
    \fi
%
   \or
    \chardef\xs@ParseState=2
    \chardef\xs@TempNum=\XeTeXcharclass`#1 %
    \let\xs@next\xs@ParseWord
    \ifx\xs@temp\xs@question
     \xs@Suffixtrue
    \else
     \ifnum\xs@TempNum>\xs@Classes
      \xs@BadChar#1{it's already a string delimiter}%
     \else
      \ifnum\xs@TempNum=0
       \xs@CreateLetter#1\xs@end
       \let\xs@next\xs@ParseWord
      \fi
     \fi
```

```
    \fi
%
   \or
    \let\xs@next\xs@ParseWord
    \chardef\xs@TempNum=\XeTeXcharclass`#1 %
    \ifx\xs@temp\xs@question
     \xs@Prefixtrue
     \chardef\xs@ParseState=3
    \else
     \ifnum\xs@TempNum>\xs@Classes
      \xs@BadChar#1{it's already a string delimiter}%
     \else
       \let\xs@next\xs@ParseWord
     \fi
    \fi
   \or
    \xs@BadChar?{it's already a string delimiter}%
   \fi
  \fi
 \fi\xs@next
 }
```

This is in case we find something we don't want in the word.

```
\def\xs@BadChar#1#2{%
 \def\xs@next##1\xs@end{}%
 \xs@BadWordtrue
 \xs@err{%
  You can't use `\noexpand#1' in `\xs@unexpanded\expandafter{\xs@TempWord}',\MessageBreak
  #2.\MessageBreak
  `\xs@unexpanded\expandafter{\xs@TempWord}' won't be searched
  }%
 }
```

In case the word is a phrase, we have to know that, so we check spaces. In case there are some, we record `word1`, then `word1 word2`, then `word1 word2 word3`, etc., as strings that may lead to phrases and should be recognized as such when XƎSearch is searching.

```
\def\xs@CheckSpaces#1\xs@end{%
 \xs@@CheckSpaces#1 \xs@end
 }
\def\xs@@CheckSpaces#1 #2\xs@end{%
 \def\xs@temp{#2}%
```

```
  \ifx\xs@temp\xs@empty
    \let\xs@next\relax
  \else
    \expandafter\xs@MakePhrase\xs@Phrase\xs@end#1\xs@end
    \def\xs@next{\xs@@CheckSpaces#2\xs@end}%
  \fi\xs@next
  }
\def\xs@MakePhrase#1\xs@end#2\xs@end{%
  \ifx\xs@Phrase\xs@empty
    \expandafter\def\csname#2@xs@phrases@\xs@cs\endcsname{}%
    \edef\xs@Phrase{#2}%
  \else
    \expandafter\def\csname#1 #2@xs@phrases@\xs@cs\endcsname{}%
    \edef\xs@Phrase{#1 #2}%
  \fi
  }%
```

In case the word was recognized as an affix, we add it to the list of affixes beginning (in the case of prefixes) or ending (in the case of suffixes) with a given letter (this is supposed to make X∃Search faster: when X∃Search scans a word, it searches e.g. prefixes if and only if there are prefixes with the same initial letter as the word under investigation, and it compares it to those words only). The affix is also added to the lists sorted by length in both orders.

```
\def\xs@GetFirstLetter#1#2\xs@end{%
  \def\xs@FirstLetter{#1}%
  }
\def\xs@MakePrefix{%
  \expandafter\ifx\csname\xs@TempWord @\xs@cs @xs@prefixes\endcsname\relax
    \expandafter\xs@GetFirstLetter\xs@TempWord\xs@end
    \ifcsname xs@prefixes@\xs@FirstLetter @\xs@cs\endcsname
      \expandafter\edef\csname xs@prefixes@\xs@FirstLetter @\xs@cs\endcsname{%
        \csname xs@prefixes@\xs@FirstLetter @\xs@cs\endcsname\xs@TempWord,}%
      \def\xs@Sign{<}%
    \xs@Insert{\xs@TempWord}{\csname xs@prefixes@\xs@FirstLetter @\xs@cs @longer\endcsname}%
      \def\xs@Sign{>}%
    \xs@Insert{\xs@TempWord}{\csname xs@prefixes@\xs@FirstLetter @\xs@cs @shorter\endcsname}%
    \else
    \expandafter\edef\csname xs@prefixes@\xs@FirstLetter @\xs@cs\endcsname{\xs@TempWord,}%
    \expandafter\edef\csname xs@prefixes@\xs@FirstLetter @\xs@cs @longer\endcsname{\xs@TempWord,}%
    \expandafter\edef\csname xs@prefixes@\xs@FirstLetter @\xs@cs @shorter\endcsname{\xs@TempWord,}%
    \fi
  \fi
```

```
  }
\def\xs@GetLastLetter#1{%
  \ifx#1\xs@end
    \let\xs@next\relax
  \else
    \let\xs@next\xs@GetLastLetter
    \def\xs@LastLetter{#1}%
  \fi\xs@next
  }
\def\xs@MakeSuffix{%
  \expandafter\ifx\csname\xs@TempWord @\xs@cs @xs@suffixes\endcsname\relax
    \expandafter\xs@GetLastLetter\xs@TempWord\xs@end
    \ifcsname xs@suffixes@\xs@LastLetter @\xs@cs\endcsname
      \expandafter\edef\csname xs@suffixes@\xs@LastLetter @\xs@cs\endcsname{%
        \csname xs@suffixes@\xs@LastLetter @\xs@cs\endcsname\xs@TempWord,}%
      \def\xs@Sign{<}%
    \xs@Insert{\xs@TempWord}{\csname xs@suffixes@\xs@LastLetter @\xs@cs @longer\endcsname}%
      \def\xs@Sign{>}%
    \xs@Insert{\xs@TempWord}{\csname xs@suffixes@\xs@LastLetter @\xs@cs @shorter\endcsname}%
    \else
     \expandafter\edef\csname xs@suffixes@\xs@LastLetter @\xs@cs\endcsname{\xs@TempWord,}%
    \expandafter\edef\csname xs@suffixes@\xs@LastLetter @\xs@cs @longer\endcsname{\xs@TempWord,}%
    \expandafter\edef\csname xs@suffixes@\xs@LastLetter @\xs@cs @shorter\endcsname{\xs@TempWord,}%
    \fi
  \fi
  }
```

These suppress the ? at the beginning or the end of the word.

Here's how we sort the list: we check each affix, and we insert the affix to be added just before the the first affix that is shorter or longer, depending on the order.

```
\def\xs@SuppressPrefix#1#2\xs@end{\def\xs@TempWord{#2}}
\def\xs@SuppressSuffix#1?{\def\xs@TempWord{#1}}
\def\xs@CountLetter#1{%
  \ifx#1\xs@end
    \let\xs@next\relax
  \else
    \advance\xs@Length1
    \let\xs@next\xs@CountLetter
  \fi\xs@next
```

```
  }
\def\xs@SortList#1,{%
  \ifx#1\xs@end
    \edef\xs@templist{\xs@templist\xs@TempAffix,}%
    \let\xs@next\relax
  \else
    \xs@Length0
    \xs@CountLetter#1\xs@end
    \ifnum\xs@Length\xs@Sign\xs@AffixLength
      \edef\xs@templist{\xs@templist\xs@TempAffix,#1,}%
      \let\xs@next\xs@EndList
    \else
      \edef\xs@templist{\xs@templist#1,}%
      \let\xs@next\xs@SortList
    \fi
  \fi\xs@next
  }
\def\xs@EndList#1\xs@end,{%
  \edef\xs@templist{\xs@templist#1}%
  }
\def\xs@Insert#1#2{%
  \def\xs@TempAffix{#1}%
  \xs@Length0
  \expandafter\xs@CountLetter#1\xs@end
  \chardef\xs@AffixLength\xs@Length
  \def\xs@templist{}%
  \expandafter\expandafter\expandafter\xs@SortList#2\xs@end,
  \expandafter\let#2\xs@templist
  }
```

Finally, we make the definition of the word. First, we associate it with the word, so we'll know which words to modify in case of a `\StopList`, and to which type it belongs (case-sensitivity, affix or full word, `!`-marked or not). Then we make both the normal replacement text and the 'no-replacement' replacement text.

```
\def\xs@PatchDef#1{%
  \expandafter\edef\csname\xs@ListName @words\endcsname{%
    \csname\xs@ListName @words\endcsname
    \xs@TempWord:::\xs@cs:::\xs@WordType:::\ifxs@Concatenate!\fi:::%
    }%
  \expandafter\ifx\csname\xs@TempWord @\xs@cs @xs@\xs@WordType\endcsname\relax%
```

```
    \xs@DefToks{\xs@FinalString}%
  \else
    \xs@DefToks\expandafter\expandafter\expandafter{%
      \csname\xs@TempWord @\xs@cs @xs@\xs@WordType\endcsname}%
  \fi
  \expandafter\ifx\csname\xs@TempWord @\xs@cs @xs@\xs@WordType @noreplace\endcsname\relax
    \xs@NoReplaceToks{}%
  \else
    \xs@NoReplaceToks\expandafter\expandafter\expandafter{%
      \csname\xs@TempWord @\xs@cs @xs@\xs@WordType @noreplace\endcsname}%
  \fi
  \ifxs@Concatenate
   \expandafter\edef\csname\xs@TempWord @\xs@cs @xs@\xs@WordType\endcsname{\the\xs@DefToks}%
    \expandafter\edef\csname\xs@TempWord @\xs@cs @xs@\xs@WordType @noreplace\endcsname{%
      \the\xs@NoReplaceToks
      \xs@unexpanded{\expandafter#1\expandafter{\xs@String}}%
      }%
  \else
    \expandafter\edef\csname\xs@TempWord @\xs@cs @xs@\xs@WordType\endcsname{%
      \noexpand\expandafter\noexpand#1\noexpand\expandafter{\the\xs@DefToks}%
      }%
  \fi
  }
```

Stopping a list is a delicate process: we have to extract the definition associated with the list from the words where it appears, and it is nested in case it is not !-marked.

```
\def\StopList{%
  \xs@ChangeCatcodes
  \xs@StopList
  }
\def\xs@StopList#1{%
  \xs@@StopList#1,\xs@end,%
  \xs@RestoreCatcodes
  }
\def\xs@@StopList#1,{%
  \def\xs@temp{#1}%
  \ifx\xs@temp\xs@end
    \let\xs@next\relax
```

```
  \else
    \ifcsname#1@xs@searchlist\endcsname
      \unless\ifcsname#1@xs@stoppedlist\endcsname
        \csname#1@xs@stoppedlist\endcsname
        \expandafter\def\expandafter\xs@ToRemove\expandafter{%
          \csname#1@xs@searchlist\endcsname
          }%
        \expandafter\expandafter\expandafter%
          \xs@PatchWords\csname #1@words\endcsname\xs@end::::::::::::%
      \fi
    \else
      \xs@err{`#1' is not a list}%
    \fi
  \let\xs@next\xs@@StopList
  \fi\xs@next
  }
```

We modify the adequate replacement text: no-replace or normal.

```
\def\xs@PatchWords#1:::#2:::#3:::#4:::{%
  \def\xs@TempWord{#1}%
  \ifx\xs@TempWord\xs@end
    \let\xs@next\relax
  \else
    \def\xs@temp{#4}%
    \ifx\xs@temp\xs@exclamation
      \expandafter\expandafter\expandafter%
     \xs@RemoveFromNoReplace\expandafter\xs@ToRemove\csname#1@#2@xs@#3@noreplace\endcsname
    \fi
    \def\xs@cs{#2}%
    \def\xs@WordType{#3}%
    \expandafter\xs@RemoveFromDef\csname#1@#2@xs@#3\endcsname
    \let\xs@next\xs@PatchWords
  \fi\xs@next
  }
```

Removing from no-replace is rather easy, since it's nothing more than:
`\expandafter\<list1-macro>\expandafter{\xs@String}`

```
\def\xs@RemoveFromNoReplace#1#2{%
  \def\xs@Erase##1\expandafter#1\expandafter##2##3\xs@end{%
    \def#2{##1##3}%
```

```
\expandafter\<list2-macro>\expandafter{\xs@String}
\expandafter\<list3-macro>\expandafter{\xs@String}
```

So we define a macro on the fly to find the definition we want to remove. If there's nothing left, we let this no-replace to `\relax`, so this word might be removed altogether when we evaluate what we find.

Normal replacement texts have the following structure:

```
\expandafter\<list1-macro>\expandafter{
 \expandafter\<list2-macro>\expandafter{
  ...
   \xs@FinalString
  ...
 }}
```

So we scan this recursively and rebuild it piecewise, removing the list that was stopped. If in the end there remains `\xs@FinalString` only, then there's no replacement text anymore, and if moreover the no-replace part is equal to `\relax`, then there's nothing left for that word and it shouldn't be tested anymore. So we let the definition associated with this word to `\relax` or we remove it from affixes.

```
    \ifx#2\xs@empty
      \let#2\relax
    \fi
    }%
  \expandafter\xs@Erase#2\xs@end
  }
\def\xs@final{\xs@FinalString}
\def\xs@TempDef{}
\def\xs@RemoveFromDef#1{%
  \def\xs@TempDef{}%
  \def\xs@Def{\xs@FinalString}%
  \unless\ifx#1\xs@final
    \expandafter\xs@Extract#1%
  \fi
  \let#1\xs@Def
  \ifx#1\xs@final
    \expandafter\ifx\csname\expandafter\xs@gobble\string#1@noreplace\endcsname\relax
      \ifx\xs@WordType\xs@words
        \let#1\relax
      \else
        \xs@RemoveFromAffixes
      \fi
    \fi
  \fi
  }
\def\xs@Extract\expandafter#1\expandafter#2{%
  \def\xs@temp{#1}%
  \unless\ifx\xs@temp\xs@ToRemove
    \edef\xs@TempDef{%
      \noexpand#1,%
      \xs@unexpanded\expandafter{\xs@TempDef}%
      }%
  \fi
  \def\xs@temp{#2}%
  \ifx\xs@temp\xs@final
```

```
    \def\xs@next{%
      \expandafter\xs@Rebuild\xs@TempDef\xs@end,%
      }%
  \else
    \def\xs@next{%
      \xs@Extract#2%
      }%
  \fi\xs@next
  }
\def\xs@Rebuild#1,{%
  \ifx#1\xs@end
    \let\xs@next\relax
  \else
    \let\xs@next\xs@Rebuild
    \edef\xs@Def{%
      \xs@unexpanded{\expandafter#1\expandafter}%
      \noexpand{%
      \xs@unexpanded\expandafter{\xs@Def}%
      \noexpand}%
      }%
  \fi\xs@next
  }%
```

Removing an affix from a list is easy: we scan each word and rebuild the list, removing the affix we want to deactivate.

```
\def\xs@RemoveFromAffixes{%
  \ifx\xs@WordType\xs@prefixes
    \expandafter\xs@GetFirstLetter\xs@TempWord\xs@end
    \let\xs@Letter\xs@FirstLetter
  \else
    \expandafter\xs@GetLastLetter\xs@TempWord\xs@end
    \let\xs@Letter\xs@LastLetter
  \fi
  \def\xs@templist{}%
  \expandafter\expandafter\expandafter%
    \xs@CleanList\csname xs@\xs@WordType @\xs@Letter @\xs@cs\endcsname\xs@end,%
  \expandafter\let\csname xs@\xs@WordType @\xs@Letter @\xs@cs\endcsname\xs@templist
  \def\xs@templist{}%
```

```
  \expandafter\expandafter\expandafter%
    \xs@CleanList\csname xs@\xs@WordType @\xs@Letter @\xs@cs @shorter\endcsname\xs@end,%
 \expandafter\let\csname xs@\xs@WordType @\xs@Letter @\xs@cs @shorter\endcsname\xs@templist
  \def\xs@templist{}%
  \expandafter\expandafter\expandafter%
    \xs@CleanList\csname xs@\xs@WordType @\xs@Letter @\xs@cs @longer\endcsname\xs@end,%
 \expandafter\let\csname xs@\xs@WordType @\xs@Letter @\xs@cs @longer\endcsname\xs@templist
  \expandafter\let\csname\xs@TempWord @\xs@cs @xs@\xs@WordType\endcsname\relax
  }
\def\xs@CleanList#1,{%
  \def\xs@temp{#1}%
  \ifx\xs@temp\xs@end
    \let\xs@next\relax
  \else
    \let\xs@next\xs@CleanList
    \unless\ifx\xs@temp\xs@TempWord
      \edef\xs@templist{\xs@templist#1,}%
    \fi
  \fi\xs@next
  }
```

### 8.4 Testing words

Here comes the big part: collecting words and testing them. When a letter follows a delimiter, we reset some values and start collecting the letters in a box...

```
\def\xs@Stack{}
\def\xs@Remainder{}
\def\xs@StartString{%
  \xs@Stringtrue
  \let\xs@StartString\relax
  \def\xs@String{}%
  \def\PrefixFound{}%
  \def\SuffixFound{}%
  \def\AffixFound{}%
  \def\xs@Stack{}%
  \def\xs@Remainder{}%
  \xs@Phrasefalse
  \setbox\xs@Box=\hbox\bgroup
```

```
  }
\let\xs@@StartString\xs@StartString
```

...and when a delimiter shows up again, unless we're tracking a phrase, we close the box, create the unknown letters that we've found in it, evaluate the word and finally output the result of this evaluation.

```
\def\xs@EndString{%
  \ifxs@String
    \egroup
    \xs@Stringfalse
    \expandafter\xs@CreateLetter\xs@PendingLetters\xs@end
    \gdef\xs@PendingLetters{}%
    \xs@Evaluate
    \xs@Restore
    \xs@StartTracing
    \expandafter\xs@Remainder
  \fi
  }
```

And here are the tests. The `F` test is for case-sensitive full words and just checks whether there is a definition for this word in this case. If it finds anything, it puts it around the string that already exists, i.e. either the bare word or the word alreay surrounded by replacement texts. Hence The bunch of `\expandafter`s. If there's a no-replace, we also add it to the existing ones. `\xs@relax` is just a placeholder to add the inhibitions defined with `\SearchOrder`.

```
\def\xs@@F@Test{%
  \expandafter\unless\expandafter\ifx\csname\xs@String @cs@xs@words\endcsname\relax
    \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter%
    \def%
    \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter%
    \xs@FinalString%
   \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter{%
      \csname\xs@String @cs@xs@words\endcsname}%
   \expandafter\unless\expandafter\ifx\csname\xs@String @cs@xs@words@noreplace\endcsname\relax
      \edef\xs@NoReplace{%
        \xs@unexpanded\expandafter{\xs@NoReplace}%
        \xs@unexpanded\expandafter{\csname\xs@String @cs@xs@words@noreplace\endcsname}%
        }%
    \fi
    \xs@Matchtrue
    \xs@relax
    \xs@relax
  \fi
  }
```

The `f` does the same thing, except it puts the word in lowercase before hand.

```
\def\xs@@f@Test{%
  \expandafter\xs@Lowercase\expandafter{\xs@String}\xs@lcString
```

```
  \expandafter\unless\expandafter\ifx\csname\xs@lcString @ncs@xs@words\endcsname\relax
    \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter%
    \def%
    \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter%
    \xs@FinalString%
    \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter{%
      \csname\xs@lcString @ncs@xs@words\endcsname}%
    \expandafter\unless\expandafter\ifx\csname\xs@lcString @ncs@xs@words@noreplace\endcsname\relax
      \edef\xs@NoReplace{%
        \xs@unexpanded\expandafter{\xs@NoReplace}%
        \xs@unexpanded\expandafter{\csname\xs@lcString @ncs@xs@words@noreplace\endcsname}%
        }%
    \fi
    \xs@Matchtrue
    \xs@relax
    \xs@relax
  \fi
  }
```

Tests on prefixes check whether there exists a prefix list beginning with the same letter as the word at stake, and in this case run the `\xs@CheckPrefixes` test.

```
\def\xs@@p@Test{%
  \xs@Affixfalse
  \expandafter\xs@GetFirstLetter\xs@lcString\xs@end
  \ifcsname xs@prefixes@\xs@FirstLetter @ncs\endcsname
    \let\xs@@String\xs@lcString
    \def\xs@cs{ncs}%
    \let\xs@WhatNext\xs@p@WhatNext
    \expandafter\expandafter\expandafter%
    \xs@CheckPrefixes\csname xs@prefixes@\xs@FirstLetter @ncs\p@order\endcsname\xs@end,%
  \fi
  \ifxs@Affix
    \xs@Affixfalse
    \xs@Matchtrue
    \xs@relax
    \xs@relax
   \fi
  }
```

```
\def\xs@@P@Test{%
  \xs@Affixfalse
  \expandafter\xs@GetFirstLetter\xs@String\xs@end
  \ifcsname xs@prefixes@\xs@FirstLetter @cs\endcsname
    \let\xs@@String\xs@String
    \def\xs@cs{cs}%
    \let\xs@WhatNext\xs@P@WhatNext
    \expandafter\expandafter\expandafter%
    \xs@CheckPrefixes\csname xs@prefixes@\xs@FirstLetter  @cs\P@order\endcsname\xs@end,%
  \fi
  \ifxs@Affix
    \xs@Affixfalse
    \xs@Matchtrue
    \xs@relax
    \xs@relax
   \fi
  }
```

Prefixes are tested one by one by creating a macro on the fly where one delimiter is the prefix. Then we put the word at stake before it and execute the macro, and if there's no first argument, then the word matches the prefix. For instance, if the word is `democracy` and the prefix is `demo` then we test
`\xs@TestPrefix democracydemo`
and obviously the first argument is empty, since `demo` is a delimiter.

```
\def\xs@CheckPrefixes#1,{%
  \def\xs@temp{#1}%
  \ifx\xs@temp\xs@end
    \let\xs@next\relax
  \else
    \def\xs@TestPrefix##1#1##2\xs@end{%
      \def\xs@temp{##1}%
      \ifx\xs@temp\xs@empty
        \xs@Affixtrue
        \def\PrefixFound{#1}%
        \def\AffixFound{#1}%
        \let\xs@next\xs@WhatNext
     \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter%
        \def%
     \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter%
        \xs@FinalString%
     \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter{%
          \csname#1@\xs@cs @xs@prefixes\endcsname}%
```

The tests for suffixes work along the same lines as those for prefixes.

```
      \expandafter\unless\expandafter\ifx\csname#1@\xs@cs @xs@prefixes@noreplace\endcsname\relax
          \edef\xs@NoReplace{%
            \xs@unexpanded\expandafter{\xs@NoReplace}%
            \xs@unexpanded\expandafter{\csname#1@\xs@cs @xs@prefixes@noreplace\endcsname}%
            }%
        \fi
      \else
        \let\xs@next\xs@CheckPrefixes
      \fi
      }%
    \expandafter\xs@TestPrefix\xs@@String#1\xs@end
  \fi\xs@next
  }
\def\xs@@S@Test{%
  \xs@Affixfalse
  \expandafter\xs@GetLastLetter\xs@String\xs@end
  \ifcsname xs@suffixes@\xs@LastLetter @cs\endcsname
    \let\xs@@String\xs@String
    \def\xs@cs{cs}%
    \let\xs@WhatNext\xs@S@WhatNext
    \expandafter\expandafter\expandafter%
    \xs@CheckSuffixes\csname xs@suffixes@\xs@LastLetter @cs\S@order\endcsname\xs@end,%
  \fi
  \ifxs@Affix
    \xs@Affixfalse
    \xs@Matchtrue
    \xs@relax
    \xs@relax
   \fi
  }
\def\xs@@s@Test{%
  \xs@Affixfalse
  \expandafter\xs@GetLastLetter\xs@lcString\xs@end
  \ifcsname xs@suffixes@\xs@LastLetter @ncs\endcsname
    \let\xs@@String\xs@lcString
```

```
    \def\xs@cs{ncs}%
    \let\xs@WhatNext\xs@s@WhatNext
    \expandafter\expandafter\expandafter%
    \xs@CheckSuffixes\csname xs@suffixes@\xs@LastLetter @ncs\s@order\endcsname\xs@end,%
  \fi
  \ifxs@Affix
    \xs@Affixfalse
    \xs@Matchtrue
    \xs@relax
    \xs@relax
   \fi
  }
\def\xs@CheckSuffixes#1,{%
  \def\xs@temp{#1}%
  \ifx\xs@temp\xs@end
    \let\xs@next\relax
  \else
    \def\xs@TestSuffix##1#1##2\xs@end{%
      \def\xs@@temp{##2}%
      \ifx\xs@temp\xs@@temp
        \xs@Affixtrue
        \def\SuffixFound{#1}%
        \def\AffixFound{#1}%
        \let\xs@next\xs@WhatNext
     \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter%
        \def%
     \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter%
        \xs@FinalString%
     \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter%
        {%
          \csname#1@\xs@cs @xs@suffixes\endcsname}%
     \expandafter\unless\expandafter\ifx\csname#1@\xs@cs @xs@suffixes@noreplace\endcsname\relax
          \edef\xs@NoReplace{%
            \xs@unexpanded\expandafter{\xs@NoReplace}%
           \xs@unexpanded\expandafter{\csname#1@\xs@cs @xs@suffixes@noreplace\endcsname}%
```

```
            }%
          \fi
        \else%
          \let\xs@next\xs@CheckSuffixes
        \fi
        }%
      \expandafter\xs@TestSuffix\xs@@String#1\xs@end
    \fi\xs@next
    }
```

## 8.5 Search order

`\SearchOrder` actually defines `\xs@Evaluate`. First it adds inhibitions to the tests, e.g. '`F!f;`' adds `\let\xs@f@Test\relax` to the `F` test in case it is positive, then it adds the tests themselves, in the specified order, to `\xs@Evaluate`.

```
\def\SearchOrder{%
  \xs@ChangeCatcodes
  \xs@SearchOrder
  }
\def\xs@SearchOrder#1{%
  \def\xs@Order{}%
  \xs@@SearchOrder#1\xs@end;%
  \edef\xs@Evaluate{%
    \xs@unexpanded{%
      \XeTeXinterchartokenstate=0
      \def\xs@NoReplace{}%
      \let\xs@FinalString\xs@String
      \expandafter\xs@Lowercase\expandafter{\xs@String}\xs@lcString
      }%
    \xs@unexpanded\expandafter{%
      \xs@Order
      \ifxs@Match
        \def\xs@next{%
          \xs@FinalString
          }%
      \else
```

If the stack is not empty, it means we're dealing with a phrase; so the evaluation is not over in case no test has succeded. We first have to test the phrase minus the last word, then

```
        \unless\ifx\xs@Stack\xs@empty
          \xs@Phrasetrue
          \expandafter\xs@PopStack\xs@Stack\xs@@end
```

the phrase minus the last two words, etc.

```
     \let\xs@next\xs@Evaluate
    \else
     \ifxs@Phrase
      \def\xs@Stack{}%
      \def\xs@next{\xs@String\xs@Restore}%
     \else
```

If the word was not a phrase, and no test was successful, we simply put the box that contains it back into the stream.

```
      \def\xs@next{\unhbox\xs@Box\xs@Restore}%
     \fi
    \fi
   \fi\xs@next
   }%
  }%
```

We initialize the tests.

```
 \let\xs@f@Test\xs@@f@Test
 \let\xs@F@Test\xs@@F@Test
 \let\xs@p@Test\xs@@p@Test
 \let\xs@P@Test\xs@@P@Test
 \let\xs@s@Test\xs@@s@Test
 \let\xs@S@Test\xs@@S@Test
 \xs@RestoreCatcodes
 }
```

This treats each specification in `\SearchOrder` and the inhibitions, if any.

```
\def\xs@@SearchOrder#1#2;{%
 \def\xs@temp{#1#2}%
 \ifx#1\xs@end
  \let\xs@next\relax
 \else
  \def\xs@Inhibit{}%
  \xs@MakeInhibit#2\xs@end
  \expandafter\expandafter\expandafter\xs@PatchTest\csname xs@@#1@Test\endcsname#1%
  \edef\xs@Order{%
   \xs@unexpanded\expandafter{\xs@Order}%
   \xs@unexpanded\expandafter{\csname xs@#1@Test\endcsname}}%
  \let\xs@next\xs@@SearchOrder
 \fi\xs@next
 }
\def\xs@MakeInhibit#1{%
```

```
  \def\xs@temp{#1}%
  \ifx#1\xs@end
    \let\xs@next\relax
  \else
    \let\xs@next\xs@MakeInhibit
    \unless\ifx\xs@temp\xs@exclamation%
      \edef\xs@Inhibit{%
        \xs@unexpanded\expandafter{\xs@Inhibit
        \expandafter\let\csname xs@#1@Test\endcsname\relax}%
        }%
    \fi
  \fi\xs@next
  }
\def\xs@PatchTest#1\xs@relax#2\xs@relax#3#4{%
  \expandafter\edef\csname xs@@#4@Test\endcsname{%
    \xs@unexpanded{#1}%
    \xs@unexpanded\expandafter{\expandafter\xs@relax\xs@Inhibit\xs@relax\fi}%
    }%
  }
```

The evaluation ends in any case with the restoration of the tests, in case they were inhibited. the remainder is the right part of a discarded phrase. For instance, if XƎSearch searches for `page layout` it will investigate `page properties` if it finds it, and the remainder is `properties`.

```
\def\xs@Restore{%
  \xs@Matchfalse
  \let\xs@f@Test\xs@@f@Test
  \let\xs@F@Test\xs@@F@Test
  \let\xs@p@Test\xs@@p@Test
  \let\xs@P@Test\xs@@P@Test
  \let\xs@s@Test\xs@@s@Test
  \let\xs@S@Test\xs@@S@Test
  \let\xs@StartString\xs@@StartString
  \edef\xs@Remainder{%
    \xs@unexpanded\expandafter{\xs@NoReplace}%
    \xs@unexpanded\expandafter{\xs@Remainder}%
    }%
  \XeTeXinterchartokenstate=1
  }
```

This is used to test phrases minus the last word on each it-

```
\def\xs@PopWord#1\xs@end#2\xs@end{%
```

eration. The stack itself is built when the beginning of a phrase is found before a natural delimiter.

To search affixes in a given order, we simply define the list to be used in tests to be the one with this order.

```
  \def\xs@String{#2}%
  \def\xs@@PopWord#2##1\xs@end{%
    \edef\xs@Remainder{##1\xs@unexpanded\expandafter{\xs@Remainder}%
      }%
    }%
  \xs@@PopWord#1\xs@end
  }
\def\xs@PopStack#1\xs@end#2\xs@@end{%
  \def\xs@Stack{#2}%
  \expandafter\xs@PopWord\xs@String\xs@end#1\xs@end
  }
\def\SortByLength#1{%
  \def\xs@temp{#1}%
  \ifx\xs@temp\xs@star
    \def\xs@AffixOrder{@shorter}%
    \let\xs@next\xs@SortByLength
  \else
    \def\xs@AffixOrder{@longer}%
    \def\xs@next{\xs@@SortByLength#1\xs@end}%
  \fi
  \xs@next}%
\def\xs@SortByLength#1{%
  \xs@@SortByLength#1\xs@end
  }
\def\xs@@SortByLength#1{%
  \ifx#1\xs@end
    \let\xs@next\relax
  \else
    \expandafter\let\csname #1@order\endcsname\xs@AffixOrder
    \let\xs@next\xs@@SortByLength
  \fi\xs@next
  }
\def\DoNotSort{%
  \def\xs@AffixOrder{}%
  \xs@SortByLength
```

```
  }
```

Searching all affixes is done by setting the `\xs@WhatNext` macro to `\xs@<affix>@WhatNext`, depending on the text being performed.

```
\def\SearchAll#1{%
  \xs@SearchAll#1\xs@end
  }
\def\xs@SearchAll#1{%
  \ifx#1\xs@end
    \let\xs@next\relax
  \else\let\xs@next\xs@SearchAll
    \if#1p%
      \let\xs@p@WhatNext\xs@CheckPrefixes
    \else
      \if#1P
        \let\xs@P@WhatNext\xs@CheckPrefixes
      \else
        \if#1s
          \let\xs@s@WhatNext\xs@CheckSuffixes
        \else
          \let\xs@S@WhatNext\xs@CheckSuffixes
        \fi
      \fi
    \fi
  \fi\xs@next
  }
\def\SearchOnlyOne#1{%
  \xs@SearchOne#1\xs@end
  }
```

Searching only one affix is simply gobbling the remaining ones in case of a successful test.

```
\def\xs@SearchOne#1{%
  \ifx#1\xs@end
    \let\xs@next\relax
  \else
    \let\xs@next\xs@SearchOne
    \expandafter\def\csname xs@#1@WhatNext\endcsname##1\xs@end,{}%
  \fi\xs@next
  }
```

### 8.6 Miscellanea

For the moment, starting and stopping the search is quite brutal.

```
\def\StopSearching{%
  \let\xs@StartString\relax
  }
\def\StartSearching{%
  \let\xs@StartString\xs@@StartString
  }
```

Patching the output very simple too.

```
\let\xs@OldOutput\relax
\def\PatchOutput{%
  \ifx\xs@OldOutput\relax
    \edef\xs@PatchOutput{%
      \noexpand\def\noexpand\xs@OldOutput{%
        \the\output
        }%
      \noexpand\output{%
        \noexpand\StopSearching
        \the\output
        \noexpand\StartSearching
        }%
      }%
    \expandafter\xs@PatchOutput
  \else
    \xs@err{Output already patched}%
  \fi
  }
\def\NormalOutput{%
  \ifx\xs@OldOutput\relax
    \xs@err{Output has not been patched}%
  \else
    \expandafter\output\expandafter{%
      \xs@OldOutput
      }%
    \let\xs@OldOutput\relax
  \fi
  }
```

As is patching the tracing.

```
\def\PatchTracing{%
  \def\xs@StopTracing{%
    \chardef\xs@tracingcommands\tracingcommands
    \chardef\xs@tracingmacros\tracingmacros
    \tracingcommands0 \tracingmacros0\relax
    }%
  \def\xs@StartTracing{%
    \tracingcommands\xs@tracingcommands
    \tracingmacros\xs@tracingmacros
    }%
  }
\def\NormalTracing{%
  \let\xs@StopTracing\relax
  \let\xs@StartTracing\relax
  }
\NormalTracing
```

finally we set everything back to normal, set some default values and say goodbye.

```
\xs@RestoreCatcodes \catcode`@=12
\SearchOrder{
  F!fPpSs;
  f!PpSs;
  P!pSs;
  p!Ss;
  S!s;
  s;
  }
\DoNotSort{pPsS}
\SearchAll{pPsS}
\XeTeXinterchartokenstate1
\endinput
```

## 8.7 A third party file for ConTEXt

This file is mostly due to Wolfgang Schuster. `\xs@contextmodule` is used when the main file is loaded to set the meaning of `\xs@unexpanded`. (ConTEXt commands have meaningful names, so I didn't want to rely on them

```
%D \module
%D   [     file=!FileName,
%D      version=!FileDate,
%D        title=\CONTEXT\ User Module,
```

as tests for ConTeXt, because there might exist commands with the same names in other formats.)

```
%D     subtitle=XeSearch,
%D       author=Paul Isambert,
%D         date=\currentdate,
%D    copyright=Paul Isambert,
%D        email=zappathustra@free.fr,
%D      license=LaTeX Project Public License]

\writestatus{loading}{ConTeXt User Module / XeSearch}
\csname xs@contextmodule\endcsname
\input xesearch.sty
\endinput
```

# 9 Index