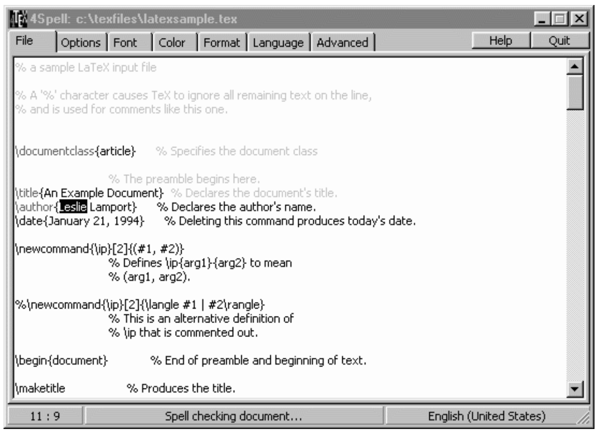
software

4Spell, a spell-checker for Windows 95/98/NT

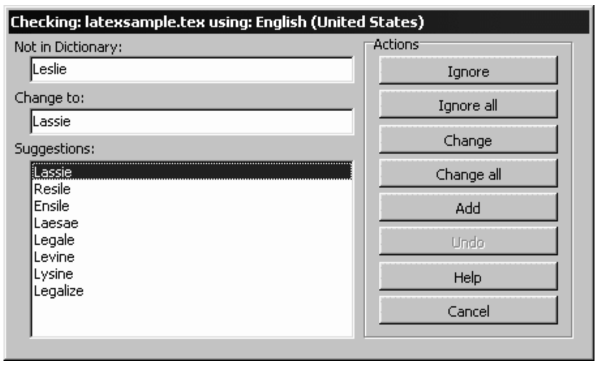
Wietse Dol and Erik Frambach

abstract
In this paper we will describe the features of 4Spell 1.1, a Windows spell-checker for TEX documents. Since there aren't many good spell-checkers around and since 4Spell only works on Windows platforms, we will also explain how the spell-checking is done. This should make it possible to write a spell-checker for other platforms (why not use perl and become platform independent :-) 4Spell is part of the new 4TEX for Windows (release expected by the end of March 1999). We realized, however, that this tool could be useful for people who do not want to use 4TEX and hence we made it a stand-alone freeware program.



```
% a sample LaTeX input file

% A '%' character causes TeX to ignore all remaining text on the line,
% and is used for comments like this one.


\documentclass{article}     % Specifies the document class

                     % The preamble begins here.
\title{An Example Document}  % Declares the document's title.
\author{Leslie Lamport}     % Declares the author's name.
\date{January 21, 1994}     % Deleting this command produces today's date.

\newcommand{\ip}[2]{(#1, #2)}
                     % Defines \ip{arg1}{arg2} to mean
                     % (arg1, arg2).

%\newcommand{\ip}[2]{\langle #1 | #2\rangle}
                     % This is an alternative definition of
                     % \ip that is commented out.

\begin{document}          % End of preamble and beginning of text.

\maketitle               % Produces the title.
```

11 : 9 | Spell checking document... | English (United States)


Checking: latexsample.tex using: English (United States)

Not in Dictionary: Leslie

Change to: Lassie

Suggestions:
Lassie
Resile
Ensile
Laesae
Legale
Levine
Lysine
Legalize

Introduction

Spell-checkers are nowadays widely used by word processors such as MS-Word and WordPerfect. They are extremely useful in correcting spelling errors, especially when writing in a non-native language.

TEX users often claim that TEX is better than those word processors, one wonders why there are so few good spell-checkers for TEX documents.

The main reason for this is that TEX documents not only contain ”normal” words, but also complex TEX commands. And TEX commands may or may not take parameters, and parameters can be delimited in any imaginable way.

Within certain TEX environments you want the words to be checked (e.g. in tables) and in others you want them to be ignored (e.g. mathematics). All in all a complex situation if you realize that TEX commands, mathematics, and normal words needn't be separated by spaces and line feeds. Any good spell-checker for TEX documents requires a TEX parser that reads the text and decides whether or not a word or a part of the word should be spell-checked. Is writing a parser difficult? The answer probably would be ”yes”, since there aren't many spell-checkers around. 4Spell proofs that writing such a spell-checker can be done and that it's not that hard to write a spell-checker that can even do more than just TEX.

4Spell features

When we started to write 4TEX for Windows we still needed the ”old” MS-Dos based AmSpell as a spell-checker. AmSpell has some serious problems/bugs when it checks your documents. We will not give you a list of those problems, but after AmSpell has checked your document you still can find spell-checking errors. This is because Amspell skips parts of your document and doesn't tell you it did.

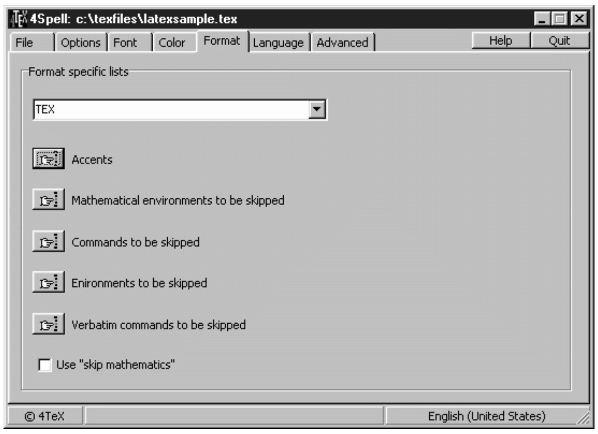
In September 1998 we had the discussion if we needed to write a spell-checker for 4TEX and concluded that it should be too time consuming to write a good program, since TEX documents are too complex. As often, complex material tends to become much simpler when you have a closer look and spend more time thinking about the structures (TEX is a structured language isn't it). When starting to write 4Spell we started not on the spell-checking routines but on describing how a TEX document should be parsed through the spell-checker. This parsing is the engine of a good spell-checker (and here AmSpell makes it's mistakes). The spell-checking routines were supplied by Aleksander Simonic. Alex is the author of WinEdt, probably the best TeX-aware shareware editor there is for OS/2 and Windows. Cooperation with Alex means that we can all benefit from the same dictionaries, which makes maintenance a lot easier.

In the next section we will describe the parsing of a document, but now we will summarize some of 4Spell's features:

- Color highlighting, i.e. actions of the spell-checker are translated into coloring of words. This makes it easy to see how your document was interpreted and (possibly) changed. Not useful you would say!? But we discovered it is extremely powerful. For instance suppose you want mathematics to be skipped by the spell-checker and you have written in your document

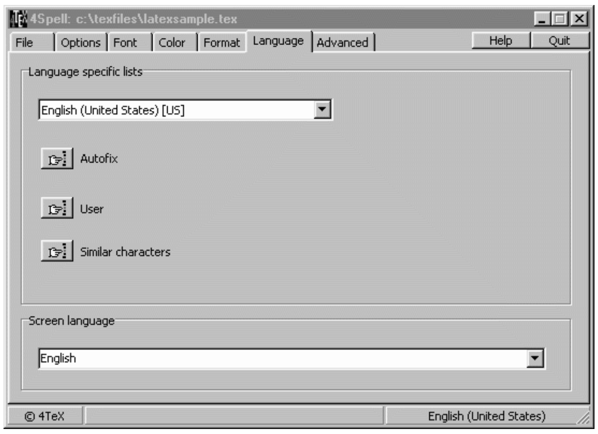
  ```
  This example $x+y will trigger probblems
  ```

  Can you predict what will happen if you check your document: it will skip the whole document after the `$xy+` since the mathematics isn't ended properly. With AmSpell (or any other spell-checker) you couldn't see this. Now you can see and solve the problem just by looking at the colored document (i.e. everything after the mathematics statement `$xy+` is colored as mathematics)!
- Support of many languages: English (American and British), German, Dutch, French, Italian, Swedish, Danish, Russian, Polish, Spanish and South-African. Since all dictionaries are plain ASCII it is simple to add your own language or to update one of the dictionaries.
- Language switching within a document allows you to write multilingual documents and spell-check all parts according to the language in which they were written.
- Generation of a word list. All the different words that are used within your document can be listed. This can be useful in deciding which words should be considered for indexing.
- Generation of a logfile, showing all actions and changes that were performed.
- Basic statistics of the document and the spell-checking run are recorded.
- Many options that can be switched on/off to increase speed and/or performance.
- You can select fonts, font sizes and character sets. This makes it possible to spell-check non-western documents (Polish, Russian, etc).
- All colors used can be changed according to your personal preferences.
- 4Spell is format dependent. It maintains specific lists for each format that you use. Lists are defined for: accents, mathematical environments, commands, environments and verbatim commands. Also mathematics (between $...$ or $$...$$) can be ignored. Note that 4Spell supports (by default) the following formats:
  – TeX and LaTeX documents
  – Rich Text Format (RTF) documents
  – plain ASCII documents
  – HTML documents
  – BibTeX documents
  Indeed, not only TeX documents can be checked and hence makes 4Spell useful not only for TeX users.
  It is easy to add a format (e.g. ConTeXt) and make the chenges to one of the format dependent lists and properties.



- 4Spell is language dependent. It maintains specific lists for each language that you use. Lists are defined for: automatic correction of typing mistakes, user specific words, and similar characters (used to specify which letters/characters are associated when looking for alternatives of an incorrect word).
- 4Spell can change its user-interface language on the fly (as 4TeX and 4Project).
- All settings for words, subwords, punctuation marks, etc., are format specific (see also the next section). This makes it easy to spell-check not only documents written in TeX, but also Plain ASCII, HTML, or RTF files. And of course you can define you own formats.
- You can check wether a word is used twice. For instance have a look at the word "one" in the next example:

  ```
  When you write very long lines and you end
  end one with a small word you tend to write
  certain words twice.
  ```



4Spell will ask you if you want to delete the second ”end” entry.

- 4Spell is lower and uppercase sensitive. For instance words as ”This”, ”tHis” and ”THis” can be changed automatically in ”This”. When checked by 4Spell it will give the suggestion ”This” for the words used in this example.

The parser

All functionality above is mostly the result of writing a (TEX) parser. To make it easier for others to write their own parser, and for those who are just curious to know how it works, we will explain the parsing algorithms.

The parser will read words until the end of a file is reached. This is done by letting a pointer start at the beginning of the file and start with the procedure `READWORD`.

STEP 1: get a word procedure READWORD

1. Skip `EndOfWord` characters until the first non-`EndOfWord` character.
2. Read and remember characters until the first `EndOfWord` character.

The result of 1 and 2. is a **word**.
This `READWORD` procedure is repeated until the end of the file. With these words you need to do a lot of checks before you can spell-check (since a word as defined above can contain (TEX) commands, etc.). Note also (within TEX) the `EndOfWord` characters are defined as: a space, a hyphen, a tilde, a Carriage-Return, a Line-Feed, and an End-Of-File character.

STEP 2: check the word for properties

For every word check the following:

1. check if the `Language Switch` (i.e., the command that is used to change dictionaries) is part of the word
2. check if one of the commands in the (TEX) `Begin Environments list` is part of the word
3. check if one of the commands in the (TEX) `commands list` is part of the word
4. check if one of the commands in the `Begin Mathematics Environments list` is part of the word
5. check if the `Mathematics Command` (e.g., \$x+y\$ or \$\$x+y\$\$) is part of the word
6. check if the `Verbatim command` is part of the word
7. check if part of the word starts a (TEX) comment (i.e. the % sign)

If one of the above is true you keep on reading words until:

1. the characters after the Language Switch command is the filename of the dictionary that should be loaded at that point.
2. the `End Environment command` is part of the word
3. the `End command command` is part of the word
4. the `End Mathematics Environment command` is part of the word
5. the `End Mathematics` is part of the word
6. the `End Verbatim character` is reached
7. end of the line is reached

This seems easy, but the problem is that when looking for, say, an environment to be ended, the same environment can start again and hence we do not stop at the first end-environment part, but at the second (or even higher) end-environment parts. This example will hopefully explain the problem:

```
\begin{skipping}
  To explain the spell problem see this example
  \begin{skipping}
    This won't work if you do not count the number
    of begin environments
  \end{skipping}
  You understand the example?
\end{skipping}
```

What the spell-checker should do is skip the complete example above. It should not stop skipping at the first `\end{skipping}` command.

When performing the actions above, we were looking for parts of the words. This means that after these actions we will have found (part of) a word preceding the action and (part of) a word after ending the action. With these two words (which may be empty) we proceed as with a word that doesn't trigger one of the actions described above.

STEP 3: divide word into subwords

Look if the word contains `SubWordPunctiationMarks`. If so, divide word into subwords.
`SubWordPunctuationMarks` are

```
,;:.!@#$&*?"%(){}[]-+=0123456789\'~^*_/|'
```

An example could clarify the meaning of subwords. Suppose we have the word

```
\def\hello{\textbf{Hello}}
```

This will be divided into four subwords:

```
\def
\hello
\textbf
Hello
```

STEP 4: check the subwords for properties

These subwords are candidates for spell-checking, but befor we spell-check these subwords we check:

1. Does the subword start with a (TEX) `Command Character` ("\"), then skip the subword (so the first three subwords of the example are skipped).
2. Is the subword one of the words in the `Ignore words list`, then ignore it.
3. Is the subword one of the words in the `Replace words list`, then replace it automatically.
4. If the subword is one of the words in the `User Dictionary`, then skip the subword.
5. If the subword is one of the words in the `Ignore Dictionary`, then skip the subword.
6. If the subword is one of the `Auto Replace word list`, then replace the subword with correct word from the `Auto Replace with word list`

If the subword doesn't belong to any of the six categories above, we spell-check the subword (Alex's routines do the job fast and easy). If the subword is correct we skip the subword. If it is not a correct word, we will search for alternatives for this (sub)word. The user will be prompted by 4Spell what to do in this case: select one of the alternatives, enter your own text, ignore this word, or add it to the user dictionary.

It seems easy, but be aware that when building a parser, you will you need to do a lot of bookkeeping, and you will need some more advanced programming tricks (e.g., all word actions and subword actions are recursive procedures).