The xpeek package*

Joel C. Salomon
⟨joelcsalomon@gmail.com⟩[†]

Released 2012/08/15

Abstract

The xpeek package provides functions for defining & managing commands which, like the familiar `\xspace`, "peek" ahead at what follows them in the input stream. They compare this token against a stored list and choose an action depending on whether there was a match.

Contents

---

* This file describes release v0.2, last revised 2012/08/15.

[†] The original code is due to Enrico "egreg" Gregorio, based on an answer he gave to a question of mine on TEX.SX; see ⟨http://tex.stackexchange.com/a/59542/2966⟩. Enrico, Joseph Wright, Bruno Le Floch, & Clemens Niederberger helped iron out the implementation; Bruno also wrote the initial version of the documentation and the near-final version of much of the code. See ⟨http://tex.stackexchange.com/q/63568/2966⟩, ⟨http://tex.stackexchange.com/q/63971/2966⟩, & ⟨http://thread.gmane.org/gmane.comp.tex.latex.latex3/2894⟩.

1 Introduction

This package provides functions for defining `\xspace`-like commands. Such commands “peek” ahead into the input stream to see what follows their invocations; their behavior varies depending on what it is that comes next.

Probably the most common LaTeX command of this sort is `\textit`. It looks ahead at the token following its argument and decides whether to insert italic correction afterward. Table 1 compares the use of `\textit` against using the font-switch command `\itshape` with & without explicit italic correction. Notice how `\textit` seems to just “do the right thing” when it comes to inserting or omitting italic correction.

`{\itshape half}hearted`	*half*hearted	(overlap)
`{\itshape half\/}hearted`	*half* hearted	(correct)
`\textit{half}hearted`	*half* hearted	(correctly including italic correction)
`by {\itshape half}.`	by *half*.	(correct)
`by {\itshape half\/}.`	by *half* .	(too wide)
`by \textit{half}.`	by *half*.	(correctly omitting italic correction)

Table 1: Automatic italic correction with `\textit`

It is, unfortunately, not hard to confuse `\textit`. As an example, consider the following code:

```
\NewDocumentCommand \naif {} {\textit{na\"\i f}}
He was a true \naif.
```

This yields

> He was a true *naïf*.

So far, so good. But using this macro in middle of a sentence may produce results the naïve user does not expect. For example, try

```
He was such a \naif he expected this example to work.
```

yielding

> He was such a *naïf*he expected this example to work.

Oops. We forgot that command words eat the spaces that follow them.

This is usually the point where TeXperts explain about following command words with explicit spaces, *e.g.*, writing “`such a \naif\ he` . . . ”, or following such commands with an empty group, “`such a \naif{} he` . . . ”, *e.g.*

Others, more pragmatically-inclined, will mention the `xspace` package. This is the sort of use-case it was developed for, after all, so let’s try it:

```
\NewDocumentCommand \naif {} {\textit{na\"\i f}\xspace}
He was such a \naif he expected this example to work.
```

This produces

> He was such a *naïf* he expected this example to work.

Even "such a *naïf*" can get the right result sometimes.

But you see, there's a kinda hitch.[1] Recall the first time we used the macro:

```
He was a true \naif.
```

Now it yields this:

> He was a true *naïf* .

Notice that ugly space between the *f* and the period? That's italic correction being added where it doesn't belong.

The trouble lies in how LaTeX implements the `\textit` logic. The macro is clever enough to look ahead, seeking for short punctuation (*e.g.*, the comma & period) which should not be preceded by italic correction. What the macro finds first, though, is that invocation of `\xspace` we added. That's not one of the "no italic correction" tokens, so the correction gets added and the sentence looks ugly.

That's where this package comes in.

(As a side note, Phillip G. Ratcliffe's `foreign` package, bundled with both MiKTeX and TeX Live and available from CTAN at ‹http://ctan.org/pkg/foreign›, is an excellent tool for this sort of foreign-word macro and was one of the inspirations for this package.)

Commands generated via the tools the `xpeek` package provides can peek ahead in the input stream, like `\textit` and `\xspace`, checking whether the next token is on their match-list. What `xpeek` adds is the ability to ignore certain tokens during that peek-ahead. These ignored tokens are not lost; the scan-ahead routine saves them and they are restored when the command executes.

2 Documentation

At this point in the development of `xpeek`, only one set of follower-dependent functions is defined. Later versions will provide tools for defining such function sets.

---

[1] Cue Malcolm Reynolds: "Don't complicate things."

2.1 Match- & Ignore-Lists

`\g_xpeek_matchlist_tl`
`\g_xpeek_ignorelist_tl`

```
\tl_gput_right:Nn \g_xpeek_⟨list⟩_tl {⟨tokens⟩}
\tl_gremove_all:Nn \g_xpeek_⟨list⟩_tl {⟨tokens⟩}
```

These two token-lists are the heart of xpeek.

The *match-list* is similar to xspace's exceptions list or LaTeX's `\nocorrlist` list for suppressing italic correction.

The functionality enabled by the *ignore-list*, on the other hand, is the *raison d'être* of this package. For example, if LaTeX's `\textit` had an ignore-list and the token `\xspace` were in that list, then the definition

```
\NewDocumentCommand \naif {} {\textit{na\"\i f}\xspace}
```

from the example above would have worked properly: `\textit` would have ignored the `\xspace`, looked past it & seen the period, and therefore omitted italic correction. When that was done, `\xspace` would execute; it too would see the period, and therefore would not insert a space.

2.2 Peek-ahead

`\xpeek_after:nw`

```
\xpeek_after:nw {⟨code⟩} ⟨token⟩
```

Similar to expl3's `\peek_after:Nw`, this sets the test variable `\l_peek_token` to ⟨*token*⟩ then executes the `{`⟨*code*⟩`}`. The ⟨*token*⟩ remains in the input stream.

2.3 Collecting Ignored Tokens

`\xpeek_collect_do:nn`

```
\xpeek_collect_do:nn {⟨ignore-list⟩} {⟨operation⟩} ⟨the rest of the input-stream⟩
```

This collects all leading tokens in the input stream which are in the ignore-list, and passes them as an argument to the operation. For example, the invocation

```
\xpeek_collect_do:nn { abc } { \foo \bar } caada
```

will collect the leading "`caa`" from the input stream and pass those tokens as an argument to the operation "`\foo \bar`". *I.e.*, the code above is equivalent to this:

```
\foo \bar { caa } da
```

(Note that it's `\bar` *within* the `{`⟨*operation*⟩`}` that gets the collected tokens as an argument.)

2.4 Searching Through Token-Lists

`\xpeek_if_in:NNTF`

`\xpeek_if_in:NNTF \⟨haystack⟩ \⟨needle⟩ {⟨true-code⟩} {⟨false-code⟩}`

Among the variants of `\tl_if_in:Nn(TF)` defined in expl3, the form `\tl_if_in:NNTF` is missing. But that's the form needed in this package, in constructions like this:

```
\xpeek_if_in:NNTF \g_xpeek_matchlist_tl \l_peek_token
  {⟨true-code⟩} {⟨false-code⟩}
```

So we define it ourselves.

3 Example

As a demonstration, here's an example of using the `xpeek` facilities to implement correct automatic italic correction *à la* `\textit`.

We set the match-list equal to LaTeX's default `\nocorrlist`; as mentioned above, `\xspace` needs to be in the ignore-list:

```
\tl_gset:Nn \g_xpeek_matchlist_tl  { , . }
\tl_gset:Nn \g_xpeek_ignorelist_tl { \xspace }
```

(Note that this does not require `\xspace` to be defined; it merely ensure that the macro will work correctly if it is defined.)

Next, we declare the new italicization command `\xit`. It calls `\xpeek_collect_do:nn` to collect any instances of `\xspace` that might follow the invocation. These are saved, but ignored for the time being.

It's within the `{`⟨*code*⟩`}` argument to `\xpeek_collect_do:nn`, between `\group_begin:` and `\group_end:`, that the italicization happens. Before we return to the default font, though, the code checks whether `\l_peek_token` is in the ignore-list; it then decides whether to include italic correction.

Recall next that the last command within the `{`⟨*code*⟩`}` is what will get the collected tokens as an argument. Since we want to keep `\xspace` if it's there, we finish up the code with `\use:n`, which simply puts its argument into the input stream.

Here's the code:

```
\NewDocumentCommand \xit {m}
  {
    \xpeek_collect_do:nn \g_xpeek_ignorelist_tl
      {
        \group_begin:
        \itshape #1
        \xpeek_if_in:NNTF \g_xpeek_matchlist_tl \l_peek_token
          { } { \/ }
        \group_end:
        \use:n
      }
  }
```

Finally, define the command we tried to make work in the introduction, and see whether we've made it work:

```
\NewDocumentCommand \naif {} {\xit{na\"\i f}\xspace}
He was a true \naif.
He was such a \naif he expected this example to work.
```

The code above yields this result:

> He was a true *naïf*. He was such a *naïf* he expected this example to work.

And work it does.

4 xpeek Implementation

```
⟨*package⟩
⟨@@=xpeek⟩
\RequirePackage{expl3, xparse}
\ProvidesExplPackage
  {xpeek} {2012/08/15} {0.2}
  {Define commands that peek ahead in the input stream}
```

4.1 Match- & Ignore-Lists

`\g_xpeek_matchlist_tl`
`\g_xpeek_ignorelist_tl`

Define the lists.

```
\tl_new:N \g_xpeek_matchlist_tl
\tl_new:N \g_xpeek_ignorelist_tl
```

(*End definition for* `\g_xpeek_matchlist_tl` *and* `\g_xpeek_ignorelist_tl` *These variables are documented on page 4.*)

4.2 Peek-ahead

`\xpeek_after:nw`
`\l__xpeek_code_tl`

Define a function (AKA "token-list") to hold the code passed in.

```
\tl_new:N \l__xpeek_code_tl
```

Store the argument in `\l__xpeek_code_tl`, then defer to `\peek_after:Nw`

```
\cs_new_protected:Npn \xpeek_after:nw #1
  {
    \tl_set:Nn \l__xpeek_code_tl {#1}
    \peek_after:Nw \l__xpeek_code_tl
  }
```

(*End definition for* `\xpeek_after:nw` *This function is documented on page 4.*)

4.3 Collecting Ignored Tokens

`\xpeek_collect_do:nn`
`\__xpeek_collect_do:nnn`
`\l__xpeek_collected_tokens_tl`

`#1` : Tokens to collect from the input stream
`#2` : The operation to be performed on the collected tokens
`#3` : The next token on the input stream (for `\__xpeek_collect_do:nnn`).

Define a list to save collected tokens in.

```
\tl_new:N \l__xpeek_collected_tokens_tl
```

Clear list and defer to auxiliary function.

```
\cs_new_protected:Npn \xpeek_collect_do:nn #1#2
  {
    \tl_clear:N \l__xpeek_collected_tokens_tl
    \__xpeek_collect_do:nnn {#1} {#2} {}
  }
```

Recursively consume tokens from the input stream so long as they match the collect-list, then apply the operation to the collected tokens. The last matching token will not yet have been collected, so pass it to the operation as well. (If there were no matching tokens, `#3` will be the empty argument that was passed by `\xpeek_collect_do:nn`.)

```
\cs_new_protected:Npn \__xpeek_collect_do:nnn #1#2#3
  {
    \xpeek_after:nw
      {
        \xpeek_if_in:NNTF #1 \l_peek_token
          {
            \tl_put_right:Nn \l__xpeek_collected_tokens_tl {#3}
            \__xpeek_collect_do:nnn {#1} {#2}
          }
          {
            #2 { \l__xpeek_collected_tokens_tl #3 }
          }
      }
  }
```

(*End definition for* `\xpeek_collect_do:nn` *This function is documented on page 4.*)

4.4 Searching Through Token-Lists

`\xpeek_if_in:NNTF`
`\l__xpeek_bool`

`#1` : Token-list to search through (the “haystack”)
`#2` : Token to search for (the “needle”)

Define an internal flag.

```
\bool_new:N \l__xpeek_bool
```

Map the “haystack”, searching for the “needle”.

```
\prg_new_protected_conditional:Npnn \xpeek_if_in:NN #1#2 { TF }
  {
    \bool_set_false:N \l__xpeek_bool
    \tl_map_inline:Nn #1
      {
        \token_if_eq_charcode:NNT #2 ##1
          {
```

```
      \bool_set_true:N \l__xpeek_bool
      \tl_map_break:
     }
   }
  \bool_if:NTF \l__xpeek_bool
   { \prg_return_true: } { \prg_return_false: }
 }
```

(*End definition for* `\xpeek_if_in:NNTF` *This function is documented on page 5.*)

```
⟨/package⟩
```

5 Test Suite

The test suite below is run automatically when this document is produced. It can also be run separately by executing

```
latex xpeek-test
```

at the command prompt.

```
⟨testsuite⟩\documentclass{article}
⟨testsuite⟩\usepackage{xparse, expl3, xpeek, qstest}
⟨testsuite⟩\begin{document}
⟨*tests⟩
```

5.1 Set-Up: Wrapping \Expect

`\ExpectIdenticalWidths`

Since the commands xpeek helps produce are not expandable, directly comparing their outputs is not feasible. Instead, typeset two versions into boxes and have qstest compare these boxes' widths. (Thanks to Heiko Oberdiek for this technique; see ‹http://tex.stackexchange.com/q/67192/2966›.)

```
\ExplSyntaxOn
\NewDocumentCommand \ExpectIdenticalWidths { m m }
 {
  \hbox_set:Nn \l_tmpa_box {#1}
  \hbox_set:Nn \l_tmpb_box {#2}
  \Expect
   * {\dim_use:N \box_wd:N \l_tmpa_box}
   * {\dim_use:N \box_wd:N \l_tmpb_box}
 }
\ExplSyntaxOff
```

(*End definition for* `\ExpectIdenticalWidths`)

5.2 Emulating \xspace

Define a simple analogue to `\xspace`.

```
\ExplSyntaxOn
\tl_const:Nn \c_xsp_exceptions_tl { ,;:.!? }
\NewDocumentCommand \xsp {}
```

```
  {
    \xpeek_collect_do:nn \c_empty_tl
      {
        \xpeek_if_in:NNTF \c_xsp_exceptions_tl \l_peek_token
          { } { ~ }
      }
  }
\ExplSyntaxOff
```

Test `\xsp`, ensuring that it is space-factor–agnostic.

```
\begin{qstest}{Emulating \xspace}{xpeek}
  \ExpectIdenticalWidths{foo bar}{foo\xsp bar}
  \ExpectIdenticalWidths{foo. bar}{foo\xsp. bar}
  \ExpectIdenticalWidths{FOO. bar}{FOO\xsp. bar}
  \ExpectIdenticalWidths{foo. bar}{foo.\xsp bar}
  \ExpectIdenticalWidths{FOO. bar}{FOO.\xsp bar}
\end{qstest}
```

```
⟨/tests⟩
⟨testsuite⟩\end{document}
```

N.B. The stand-alone test-suite will not produce any output, only a log file.

Change History

Version 0.1
  General: First useful version 1

Version 0.2
  General: Added test-suite 8

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.