Macro interfaces and the *getoptk* package

Michael Le Barbier Grünewald

1 Introduction

We present the *getoptk* macro package for the *plain* format. It eases the definition of macros whose interface is similar to the one used by TeX primitives such as `\vrule` or `\hbox`. We discuss some characteristics of interface styles and a short classification of these before describing our package. The implementation of the `\readblanks` macro, a variant of the primitive `\ignorespaces` triggering a callback, seems to us especially interesting.

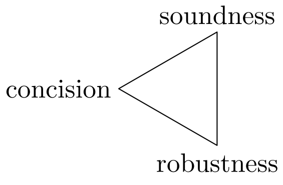
The *getoptk* macro package is similar to *xkeyval*, in that it allows optional arguments to be specified as a dictionary. However, it avoids the introduction of a new syntactic construction for the concrete form of the dictionary. Instead, it tries to imitate the convention used by `\vrule` and similar TeX primitives.

2 Literate programming

We use Norman Ramsey's NOWEB [5] literate programming tool to present our code. A file is split up in *chunks*, each of which is given an identifier that we will always write in italics; for example, *Definition of getoptk*. In this text, chunk names that are not also file names are capitalised; although this is not conventional English syntax, it helps recognising a chunk name as such in the text.

3 Characteristics of interfaces

The main characteristics of macro interfaces are organised around the three ideas of *concision, robustness* and *soundness*, which in turn are the three vertices of the following *tension triangle:*



The idea of concision expresses itself in interfaces encouraging a terse and short way to type in a manuscript. Robustness is the ability of an interface to perform well in a wide range of contexts, especially nested calls. Sound interfaces mix well in the manuscript and do not break its homogeneity. They are easy to memorise and help in having a nice looking manuscript.

These three ideas are distinct: The `\proclaim` macro in *plain TeX* and the modal behaviour of the *letter* example format described in *The TeXbook* [3] both put an emphasis on concision but break robustness and soundness. The `verbatim` environment in LaTeX is sound and concise but not robust. If robustness is needed in a verbatim typesetting job, we may rely on the usual markup and named characters to input our data: we get a sound and robust answer to our problem, but this lacks concision.

Remarks:

1. Lexical analysis techniques can be used to build concise macro interfaces, often at the price of robustness. This loss is easily circumvented by cleanly separating the lexical analysis part and the processing part of the macro job.
2. Concise interfaces help in producing a text that is easy to maintain. The maintainability burden caused by the lack of concision of an interface can sometimes be ameliorated by a third party tool — *e.g.*, the code pretty-printer included in the literate programming tool WEAVE [4] — used to automatically generate parts of the text using a non-concise macro set.
3. Sound interfaces make life easier for third party software processing TeX manuscripts.

4 Bestiary

Let us quickly review different styles of familiar interfaces and means to define macros using them. Note that many macros do not use purely one of the styles of interface described below, but rather a combination of them.

4.1 Simple

In the *simple* interface style, the control sequence is followed by its arguments, each one being either a token or a group. Macros using the simple interface style are defined by the `\def` primitive:

```
\def\example#1#2{%
 This replacement text uses #1 and #2.
}
```

4.2 Delimited

Macros using the *delimited* interface style take advantage of the ability of `\def` to be given somewhat arbitrary argument delimiters. This feature lets the macro usage blend smoothly in the surrounding text. A popular example is the `\proclaim` macro defined in *plain*:

```
\proclaim Theorem 1. {\TeX} has a powerful
 macro capability.\par
```

This macro has two delimited arguments, one starting right after the `\proclaim` control sequence and running to the first dot on the line, the second starting after the point and ending with the current paragraph, signalled by a double carriage return or an explicit `\par` as in the example above. If `\proclaim` were designed to use the simple interface style, the previous usage example would have look like this:

```
\proclaim{Theorem 1}{{\TeX} has a powerful
 macro capability.}
```

4.3 Register

The *register* interface style relies on registers and control sequences instead of formal arguments to pass informations to the macro. With a hypothetical implementation of `\proclaim` using this interface style, the previous example could be:

```
\def\proclaimlabel{Theorem 1}
\def\proclaimtext{{\TeX} has a powerful
 macro capability.}
\proclaim
```

This use of *global variables*, as it is often referred to in classical programming languages, usually breaks the ability of a macro to support nested calls. This is not always a problem in TeX, where modifications of registers can be made local to a group. There is a realm where this use of global variables is often the rule: machine level programming. Indeed, many BIOS or OS functions on PCs are serviced through software interruptions. In the typical case, the registers of the machine are assigned values corresponding to the parameters of the call, and the interruption is then triggered.

Some interactions with the typesetting engine TeX are achieved through the use of dedicated registers. Using a register style for the interface of a macro may give the user a feeling he is interacting with TeX as a machine. This style may be appropriate for font selection schemes and other "system services". The main macro of our package partially uses this interface style.

There is no special provision needed to define a macro using such an interface, though some packages, including *getoptk*, provide the user with specialised macros used to set the values of the registers.

4.4 Keyword

The TeX primitives `\hrule`, `\vrule`, `\hbox`, `\vbox` and `\vtop` use a special interface style that we call the *keyword* interface style. A typical call to the `\hrule` primitive is:

```
\hrule width 12pt depth 2pt height 10pt
```

Each parameter to the call is introduced by a keyword, then comes the actual parameter associated to the keyword. Keywords have no fixed order and it is possible to repeat the same keyword more than once or to omit some or all of them. It is a very flexible way to pass arguments to macros, similar to *labels* in the OCaml programming language. Unluckily, there is no facility in TeX itself or in *plain* to define macros using this interface style. The second part of this paper presents such a facility. A close variant of this style is the *keyval* style discussed hereafter and whose popularity among LaTeX hackers is increasing.

4.5 Starred

Macros having a *starred* variant are well known to LaTeX users. Structure domain related macros, such as `\chapter` or `\section`, usually have a starred variant whose behaviour is similar to the original version but does not produce an entry in the table of contents of the document or receive a section number. The use of a macro using this interface style and the simple interface style is illustrated by the following line of code:

```
\section*{Introduction}
```

Starred variants of macros are supported by the preferred LaTeX methods for creating new macros. Any macro defined by `\newcommand` can use the pseudo-predicate `\@ifstar` to check for itself being called with a star or not.

4.6 Bracket

A common feature found in interfaces to macros defined in LaTeX is the use of a bracketed optional argument. We call this the *bracket* style interface. The `\cite` macro defined by LaTeX uses this interface style and the simple one, as illustrated by:

```
\cite[Theorem~1]{TEXBOOK}
```

The definition of macros using this style of interface is supported in LaTeX by `\newcommand`, where the `\cite` command used above could have been (but was not) defined by

```
\newcommand{\cite}[][1]{...}
```

4.7 Keyval

The *keyval* interface style is named after a popular LaTeX package *keyval* by David Carlisle [2] and its successor *xkeyval* by Hendri Adriaens [1].

Macros using this interface style allow options of the form `key=value`; a sample use is:

```
\mybox[text=red,left=5pt]{some text}
```

It is easy to define macros using this interface style with the *xkeyval* package, which is available to *plain* and LaTeX users. This interface is probably unsound in a *plain* TeX document but may fit well in a LaTeX document, since the notation it uses is reminiscent of the one used for package arguments.

5 Presentation of the package

We now describe the interface and the implementation of the *getoptk* package. Our goal is to provide users of *getoptk* with an easy way to define macros using the *keyword interface style* described above. This style is a very flexible way to pass arguments to macros and already used by TeX primitives. The use of this style therefore favours *soundness* of the interface.

Rather than presenting a formal specification of our macros, let us take a look at an example of utilisation and use that as a basis to discuss the features and the operation of the package.

5.1 Usage example

We show how to use *getoptk* to define `\example`, a macro having a mandatory argument and accepting optional arguments in the *keyword* interface style.

⟨*example.tex*⟩ ≡
```
  \input getoptk
  \catcode`\@=11
  ⟨Dictionary definition⟩
  ⟨Main definition⟩
  ⟨Usage⟩
  ⟨Usage equivalence⟩
  \bye
```

We need first to read the *getoptk* package. In the code chunk *Dictionary definition* we require the creation of a new optional argument dictionary that we fill with bindings between keywords and behaviours. In the following, we refer to a dictionary of this kind as a *behaviour dictionary*. We then define the `\example` command itself where the magic happens in *Main definition* and add a few examples in *Usage* and their replacement text after the call to `\getoptk` in *Usage equivalence*.

Note that we use private names containing `@` in this example, thus the example starts with the familiar `\catcode` mantra.

We require the creation of a fresh behaviour dictionary — a data structure represented by a control sequence — with `\newgetoptkdictionary` to bind keywords to behaviours. In this example, the binding operations are performed by the commands

`\defgetoptkflag`,
`\defgetoptkbracket` and
`\defgetoptkcount`.

The bindings are written in the dictionary associated with `example` because it is the last one created. (It is possible to choose another dictionary with `\setgetoptkdictionary`.) The binding macros mix the register interface style and the simple one. This is convenient because this avoids repeating the argument `example` each time a binding macro is called and there is no plausible usage scenario involving nasty nested calls.

⟨*Dictionary definition*⟩ ≡
```
  \newgetoptkdictionary{example}
  \defgetoptkflag{alpha}{(alpha)}
  \defgetoptkflag{beta}{(beta)}
  \defgetoptkcount{gamma}{(gamma #1)}
  \defgetoptkbracket{delta}{%
    \ifgetoptkbracket
      (delta "#1")%
    \else
      (delta)%
    \fi}
```

Each binding macro has two arguments: a *keyword*, that consists of a sequence of catcode 11 tokens, and a *behaviour*, a valid replacement text for a macro. The binding macro arranges things so that each occurrence of the keyword seen in the call to `\example` triggers the evaluation of the corresponding behaviour. Before we give more details on this triggering mechanism and the semantics of the binding, let us look at the definition of `\example`:

⟨*Main definition*⟩ ≡
```
  \def\example{%
    \setgetoptkdictionary{example}%
    \getoptk\example@M}
  \def\example@M#1#2{%
    \par\noindent[{\tt #1}][#2]}
```

We see that the definition of `\example` is basically a call to `\example@M`, supervised by `\getoptk`. The task of `\getoptk` is to look for keywords on the input stream and aggregate the corresponding behaviours and arguments. The resulting aggregate is then given as the first argument to `\example@M`. Before we pass control to `\getoptk`, we first use `\setgetoptkdictionary` to activate the behaviour dictionary defined above.

Returning to the above *Dictionary definition*, the two calls to `\defgetoptkflag` bind the keywords `alpha` and `beta` with the behaviours `(alpha)` and `(beta)`. These are saved as the replacement texts of `\getoptk@behaviour@example@alpha` and `\getoptk@behaviour@example@beta`. Given this, `\getoptk` arranges things so that the sequence:

⟨*Usage*⟩ ≡
```
  \example beta alpha {omega}
```

expands to:

⟨*Usage equivalence*⟩ ≡
```
  \example@M{%
    \getoptk@behaviour@example@beta
    \getoptk@behaviour@example@alpha
  }{omega}
```

The call to `\defgetoptkcount` binds `gamma` to the behaviour `(gamma #1)` but also notes that the `gamma` keyword must be followed by an integer — a valid right-hand-side for *count* registers. This integer will replace the formal paragraph `#1` when behaviours are triggered.

The last binding of our example is performed by `\defgetoptkbracket`, that defines a keyword admitting an optional bracketed argument. As illustrated by our example, the behaviour uses the predicate `\ifgetoptkbracket` to test for the presence of an optional argument. This is a true predicate created by the `\newif` command. The sequence

⟨*Usage*⟩ +≡
```
  \example gamma 2 delta [10] {omega}
```
then expands to

⟨*Usage equivalence*⟩ +≡
```
  \example@M{%
    \getoptk@behaviour@example@gamma{2}%
    \getoptkbrackettrue
    \getoptk@behaviour@example@delta{10}%
  }{omega}
```
The `\getoptk` command is generous in accepting white space. In the following example, both calls to `\example` are expanded the same way.
```
\example delta[2]gamma10beta{omega}
\example delta [2] gamma
 10 beta {omega}
```
The *getoptk* package provides more binding macros, reading dimensions or tokens, and it is also possible to create new ones (6.6).

5.2 Criticism of the interface

We criticise the interface of a macro using `\getoptk` to get its optional arguments, in view of the three characteristics we isolated in the introduction:

soundness holds, because the interface mimics the behaviour of some TeX primitives;

concision is as respected as it can, but the interface to a macro admitting a large number of optional arguments cannot be that concise;

robustness seems to hold, and it is also possible to circumvent the direct use of `\getoptk` and directly construct the resulting call, as demonstrated by the *Usage equivalence* chunks above.

6 Implementation

We dive here into the deepest part of the job.

6.1 Overview

There are three important parts in the implementation. The *Definition of getoptk* and the elaboration of dedicated *Lexical analysis procedures* will almost entirely capture our attention. It is also useful to define macros manipulating *Behaviour dictionaries*: the techniques used there are very similar to those used in list processing [3, p. 378] and we will not give many details. In these three parts, there are a few short macro definitions that may have a general usefulness, we gather them in *Ancillary definitions.*

⟨*getoptk.tex*⟩ ≡
```
  \catcode`\@=11
  ⟨Ancillary definitions⟩
  ⟨Lexical analysis procedures⟩
  ⟨Definition of getoptk⟩
  ⟨Behaviour dictionaries⟩
  \catcode`\@=12
```

The very special nature of TeX programs forbids the use of literate programming techniques to describe the flow of the procedure. We use instead an imperative pseudo-code notation, where d stands for the active behaviour dictionary and c is the argument given to `\getoptk`, the callback taking control of the execution flow after `\getoptk` completes its task.

Algorithm 1 Workflow of *getoptk*
```
$x \leftarrow \emptyset$ {accumulator}
$f \leftarrow true$ {loop flag}
while f do
  $k \leftarrow$⟨incoming tokens⟩
  if $k$ is bound in $d$ then
    $b \leftarrow$ behaviour of $k$
    $a \leftarrow$ argument of $k$
    stack $b$ and $a$ on $x$
  else
    $f \leftarrow false$
    apply $c$ to $x$
    process tokens in $k$
  end if
end while
```

The work needed to implement this simple procedure falls in three categories. First, we have to manipulate data structures. The easiest way to do this is to use registers to store arguments and output of procedures manipulating the data structure. There is no special difficulty in this lengthy task. Second, structured programming in TeX is always an involved task, since the decision parts of the code have to put the bits whose processing they require on the stream of incoming tokens. We end up with many short macros mutually calling themselves. Again, there is no real difficulty here but rather a code organisation problem. Third, there are a few tricks in the lexical analysis techniques used (6.7).

6.2 Ancillary definitions

We use many short macros whose definition can be found in *The TEXbook*, such as `\gobble`:

⟨*Ancillary definitions*⟩ ≡
```
  \def\gobble#1{}
```
  ⟨*More ancillary definitions*⟩

We also need `\cslet`, `\elet` and `\csdef` but omit the definition of these classical macros here. Instead we proceed to the definition of `\tokscat` used later in the text to concatenate two token registers into a third, as in

```
\tokscat\toks0 &\toks2\to{\toks0}
```

Note the space after the first occurrence of the character `0`, it is mandatory to put a space there if you do not use a named token register.

⟨*Ancillary definitions*⟩ +≡
```
  \def\toksloadcsexpansion#1\to#2{%
    #2=\expandafter{#1}}
  \def\tokscat#1&#2\to#3{%
    \beginnext
    \edef\tokscat@a{\the#1\the#2}%
    \toks2={#3}%
    \toksloadcsexpansion\tokscat@a
      \to{\toks4}%
    \edef\next{\the\toks2={\the\toks4}}%
    \endnext}
```

This control sequence is similar to `\concatenate` [3, p. 378] concatenating two lists.

6.3 Description of behaviour dictionaries

A behaviour dictionary is a list of triples represented like this:

`\\{{`⟨*keyword*⟩`}{`⟨*parser*⟩`}{`⟨*behaviour*⟩`}}`

We already discussed the *keyword* and *behaviour* fields (5.1) but there is a new feature. The *parser* field contains a control sequence whose job is to read the argument associated with *keyword,* removing its tokens from the input stream and storing them in a dedicated token register:

⟨*Definition of getoptk*⟩ ≡
```
  \newtoks\getoptkargument
```

Once the parser has completed its task, it gives control back to *getoptk* by calling `\getoptkcallback`.

The `\getoptk` macro requires that a valid behaviour dictionary be stored in `\getoptkdictionary` before it is called. The `\setgetoptkdictionary` macro can be used for this; it is defined in *Behaviour dictionaries*, together with macros used in *Dictionary definition* from the usage example (5.1).

6.4 Definition of entry and exit blocks

The main piece of code is divided into many small macros, whose names consist of a common prefix *getoptk* followed by the private namespace character `@` and a capital letter. This notation is inspired by assembly languages providing local labels (usually denoted by a single digit). Using this notation puts the emphasis on all these macros being private pieces of a larger entity. The letter is sometimes chosen according to the function of the code (continue, end or exit, loop, main, predicate) but most of the time, letters are simply used in sequence, from A to Z. Small letters are used for private variables. Here is the, somewhat deceiving, definition of `\getoptk`:

⟨*Definition of getoptk*⟩ +≡
```
  \def\getoptk#1{%
    \beginnext
    \toks0={#1}%
    \toks2={}%
    \toks4={}%
    \toks6={}%
    \getoptkargument={}%
    \getoptk@L}
```

The argument of `\getoptk` is a callback, it is saved in `\toks0`, that corresponds to c in Algorithm 1. The content of `\getoptkargument` and some scratch registers are erased. The register `\toks2` plays the role of the accumulator x. The first token of the replacement text is `\beginnext`, which has not yet been defined. As its name suggests, it has a companion macro `\endnext`:

⟨*Ancillary definitions*⟩ +≡
```
  \def\beginnext{%
    \begingroup
    \let\next\undefined}
  \def\endnext{%
    \expandafter\endgroup\next}
```

This kind of construction is familiar to TEX programmers using `\edef` constructs: it allows the easy opening of a group inside which we are allowed to play all kinds of register-based games and finally use `\edef` to compute an appropriate replacement text for `\next`. The exit block of our procedure is:

⟨*Definition of getoptk*⟩ +≡
```
  \def\getoptk@E{%
    \edef\next{%
      \the\toks0{\the\toks2}%
      \the\toks4}%
    \endnext}
```

We already know that `\toks0` holds the callback registered by the user who called `\getoptk`, and `\toks2` the material gathered so far by the whole procedure.

Register `\toks4` corresponds to k in Algorithm 1 and contains tokens that were removed from the input stream but failed to compare with a keyword bound in the active behaviour dictionary. Please take a look again at the first example of usage discussed above (5.1):

```
\example beta alpha {omega}
```

When the `\getoptk` procedure completes it ultimately calls `\getoptk@E`. Right after the evaluation of `\edef` the replacement text of `\next` is

```
\example@M{%
  \getoptk@behaviour@example@beta
  \getoptk@behaviour@example@alpha}
```

which `\expandafter` puts again in the stream of incoming tokens, therefore replacing the original sequence `\example beta alpha`.

6.5 Definition of the main loop

We read the incoming tokens that may stand for a keyword. For this, we use two custom lexical analysis procedures (6.7): `\readblanks` that discards blanks on the input stream and `\readletters` that gathers tokens with *catcode* = 11 in a register.

⟨*Definition of getoptk*⟩ +≡

```
  \def\getoptk@L{%
    \readblanks\then\getoptk@A\done}
  \def\getoptk@A{%
    \readletters\to\toks4\then
      \getoptk@B
    \done}
```

We now look for the keyword stored in `\toks4` in the dictionary `\getoptkdictionary`, using the classical list scanning technique described in *The TEXbook* [3, p. 378].

⟨*Definition of getoptk*⟩ +≡

```
  \def\getoptk@B{%
    \let\getoptk@N\getoptk@E
    \let\\\getoptk@S
    \getoptkdictionary
    \getoptk@N}
```

The scanning macro `\getopk@S` first unpacks its argument into a triple

\\{{⟨*keyword*⟩}{⟨*parser*⟩}{⟨*behaviour*⟩}}

The real work happens in `\getoptk@T`, which sets the value of the `\getoptk@N` callback to a value requiring the lecture of an argument to *keyword* with *parser*. Two values for the *parser* field have a special meaning: an empty value means no argument, while `[]` means a bracketed optional argument.

⟨*Definition of getoptk*⟩ +≡

```
  \def\getoptk@S#1{\getoptk@T#1}
  \def\getoptk@T#1#2#3{%
    \edef\getoptk@a{\the\toks4}%
    \def\getoptk@b{#1}%
    \def\getoptk@p{#2}%
    \ifx\getoptk@a\getoptk@b
      \let\\\gobble
      \toks6={}%
      \toks8={#3}%
      \def\getoptk@N{#2}%
      \def\getoptk@a{}%
      \ifx\getoptk@p\getoptk@a
        \let\getoptk@N\getoptkcallback
      \fi
      \def\getoptk@a{[]}%
      \ifx\getoptk@p\getoptk@a
        \let\getoptk@N\getoptk@O
      \fi
    \fi}
```

Each parser must end with a call to the callback `\getoptkcallback` that is in charge of aggregating behaviours and their arguments in the accumulator `\toks2`. It relies on `\tokscat` to concatenate two token registers. The `\getopk@O` parser uses the register `\toks6` to communicate the position of `\ifgetoptkbracket` to `\getoptkcallback`. The old value of `\getoptk@p` defined in `\getoptk@T` is used to recognise the case when we did not have to gather an argument. Ultimately, we branch to the main loop `\getoptk@L` again.

⟨*Definition of getoptk*⟩ +≡

```
  \def\getoptkcallback{%
    \tokscat\toks2 &\toks6\to{\toks2}%
    \tokscat\toks2 &\toks8\to{\toks2}%
    \def\getoptk@a{}%
    \ifx\getoptk@p\getoptk@a
      \toks6={}%
    \else
      \edef\getoptk@N{%
        \toks6={%
          {\the\getoptkargument}%
        }%
      }%
      \getoptk@N
    \fi
    \tokscat\toks2 &\toks6\to{\toks2}%
    \getoptk@L}
```

6.6 Definition of parsers

The final step is the definition of parsers. Here is the one associated with keywords admitting an optional bracketed argument.

⟨*Definition of getoptk*⟩ +≡

```
  \newif\ifgetoptkbracket
  \def\getoptk@O{%
    \readblanks\then
      \futurelet\getoptk@t\getoptk@P
    \done}
  \def\getoptk@P{%
    \ifx\getoptk@t[%]
      \toks6={\getoptkbrackettrue}%
      \let\getoptk@N\getoptk@Q
    \else
      \toks6={\getoptkbracketfalse}%
      \let\getoptk@N\getoptkcallback
   \fi
   \getoptk@N}
  \def\getoptk@Q[#1]{%
    \getoptkargument={#1}%
    \getoptkcallback}
```

We next define a meta-parser and use it to define a parser for integers. This meta-parser gives access to internal TEX parsers associated with the various types of registers: to parse a value that is a valid right-hand-side for a *dimen* register, it must be provided a scratch *dimen* register, and so on. The value of this register is modified within a group, so any register is suitable as an argument for the meta-parser.

⟨*Definition of getoptk*⟩ +≡

```
  \def\getoptkmetaparser#1{%
    \def\getoptkmetaparser@r{#1}%
    \afterassignment\getoptkmetaparser@A
    #1}
  \def\getoptkmetaparser@A{%
    \beginnext
    \toks2=\expandafter{%
      \getoptkmetaparser@r
    }%
    \edef\next{%
      \noexpand
      \getoptkmetaparser@B{\the\toks2}%
    }%
    \endnext}
  \def\getoptkmetaparser@B#1{%
    \edef\getoptk@N{%
      \getoptkargument={\the#1}%
    }%
    \getoptk@N
    \getoptkcallback}
  \def\getoptkcountparser{%
    \getoptkmetaparser{\count0 }}
```

⟨*Definition of getoptk*⟩ +≡
  ⟨*More parsers*⟩

We similarly define

`\getoptkdimenparser`,
`\getoptkskipparser` and
`\getoptktoksparser`

in *More parsers*, but omit these details.

It is possible to define new parsers by following the pattern of `\getoptcallback`: use a `\beginnext`/`\endnext` pair to read the *argument* and then arrange for `\next` to expand to

`\getoptkargument={`*argument*`}\getoptkcallback.`

6.7 Lexical analysis procedures

We define two macros performing a simple task related to lexical analysis. The first one, `\readblanks`, has the seemingly simple task of discarding blank tokens on the input stream and triggering a callback when it finds the first non-blank token. The second one, `\readletters`, gathers in a register the largest prefix of catcode 11 tokens found in the input stream, and triggers a callback.

The TEX primitive `\ignorespaces` does not support any callback. Thus we have to implement a macro of our own achieving this effect. It is not as easy as it seems, though at a high level, the task looks straightforward. We denote our callback by c:

Algorithm 2 Reading white space

```
  $f \leftarrow true$
  while $f = true$ do
    $t \leftarrow$ incoming token
    if $t$ is a space or a newline token then
      discard $t$
    else
      $f \leftarrow false$
    end if
  end while
  trigger $c$
```

We use `\futurelet` to scan incoming tokens and therefore need to bind a space token and a newline token to some control sequences. We can then apply `\ifx` to compare the incoming token and these control sequences. Binding these tokens to control sequences cannot be done with a simple `\let` as they would then be overlooked by TEX. But a clever use of `\futurelet` will do:

⟨*Lexical analysis procedures*⟩ ≡

```
  \begingroup
  \catcode`\*=13
  \def*#1{}
  \global\futurelet\spacetoken*^^20\relax
  \global\futurelet\newlinetoken*^^0a\relax
  \endgroup
```

The main loop uses `\futurelet` to get an incoming token and test it in turn against `\spacetoken`, `\newlinetoken`, `\par` and `\input` to make the decision of exiting the loop with `\readblanks@E`, discarding a blank with `\readblanks@S` or a paragraph with

`\readblanks@I`, or expanding an `\input` command with `\readblanks@X`.

⟨*Lexical analysis procedures*⟩ +≡
```
  \def\readblanks\then#1\done{%
    \beginnext
    \def\next{#1}%
    \readblanks@L}
  \def\readblanks@L{%
    \futurelet\readblanks@t\readblanks@A}
  \def\readblanks@A{%
    \let\readblanks@N\readblanks@E
    \ifx\readblanks@t\spacetoken
      \let\readblanks@N\readblanks@S
    \fi
    \ifx\readblanks@t\newlinetoken
      \let\readblanks@N\readblanks@S
    \fi
    \ifx\readblanks@t\par
      \let\readblanks@N\readblanks@I
    \fi
    \ifx\readblanks@t\input
      \let\readblanks@N\readblanks@X
    \fi
    \readblanks@N}
  \def\readblanks@E{\endnext}
```

The actual discard of a space token requires a small trick. An easy way to discard a general token is to use a macro ignoring its argument: this will not work here, because space tokens are ignored by TeX as it searches the input stream for a macro argument. An assignment to a counter register will consume a space token following it: the space we want to get rid of then marks the end of a numeric constant and is discarded. We use `\afterassignment` to regain control after this.

⟨*Lexical analysis procedures*⟩ +≡
```
  \long\def\readblanks@S{%
    \afterassignment\readblanks@L
    \count0=0}
```

The two last choices, *ignore* and *expand*, are readily implemented:

⟨*Lexical analysis procedures*⟩ +≡
```
  \def\readblanks@I#1{%
    \readblanks@L}
  \def\readblanks@X{%
    \expandafter\readblanks@L}
```

⟨*Lexical analysis procedures*⟩ +≡
  ⟨*Definition of readletters*⟩

Our second analysis procedure `\readletters` gathers tokens with catcode 11 in a register and triggers a callback. It is much like `\readblanks`, a `\futurelet`-based loop. We will not reproduce it here.

7 Conclusion

We've surveyed macro interface styles and implemented the interface used by `\vrule`, etc., for plain TeX. We hope this will be of use to other macro writers.

◇ Michael Le Barbier Grünewald
Hausdorff Center for Mathematics
Villa Maria Endenicher Allee 62
D 53 115 Bonn
Germany
`michi (at) mpim-bonn dot mpg dot de`

References

[1] Hendri Adriaens, *The xkeyval package.* 2008. `http://mirror.ctan.org/macros/latex/contrib/xkeyval`

[2] David Carlisle, *The keyval package.* 1999. `http://mirror.ctan.org/macros/latex/required/graphics`

[3] Donald E. Knuth, *The TeXbook.* Addison Wesley, Massachusetts. Corrected edition, 1996.

[4] Donald E. Knuth, *The Web System of Structured Documentation.* 1983.

[5] Norman Ramsey, *Noweb: A Simple, Extensible Tool for Literate Programming.* `http://www.cs.tufts.edu/~nr/noweb`