The lparse package

Josef Friedrich
josef@friedrich.rocks
github.com/Josef-Friedrich/lparse

0.3.0 from 2025/07/02

```
\def\test{\par\directlua{
  local oarg, star, marg = lparse.scan('o s m')
  tex.print('o: ' .. tostring(oarg))
  tex.print('s: ' .. tostring(star))
  tex.print('m: ' .. tostring(marg))
}}

\test{marg} % o: nil s: false m: marg
\test[oarg]{marg} % o: oarg s: false m: marg
\test[oarg]*{marg} % o: oarg s: true m: marg
```

Contents

1 Introduction

The name `lparse` is derived from `xparse`. The `x` has been replaced by `l` because this package only works with LuaTEX. `l` stands for *Lua*. Just as with `xparse`, it is possible to use a special syntax consisting of single letters to express the arguments of a macro. However, `lparse` is able to read arguments regardless of the macro systemd used - whether LATEX or ConTEXt or even plain TEX. Of course, LuaTEX must always be used as the engine.

Similar projects

For ConTEXt there is a similar argument scanner (see ConTEXt Lua Document cld-mkiv). This scanner is implemented in the following files: toks-scn.lua toks-aux.lua toks-ini.lua ConTEXt scanner apparently uses the token library of the LuaTeX successor project luametaTEX: lmttokenlib.c

2 The Lua API of `lparse`

2.1 Description of the argument specification

The following lists describing the argument types are taken from the manuals usrguide and xparse. The descriptive texts of the individual argument types have only been slightly adjusted. The argument types that are not yet supported are bracketed.

- m A standard mandatory argument, which can either be a single token alone or multiple tokens surrounded by curly braces `{}`. Regardless of the input, the argument will be passed to the internal code without the outer braces. This is the lparse type specifier for a normal TEX argument.
- r Given as `r`⟨*token1*⟩⟨*token2*⟩, this denotes a "required" delimited argument, where the delimiters are ⟨*token1*⟩ and ⟨*token2*⟩. If the opening delimiter ⟨*token1*⟩ is missing, `nil` will be returned after a suitable error.
- R Given as `R`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`, this is a "required" delimited argument as for `r`, but it has a user-definable recovery ⟨*default*⟩ instead of `nil`.
- v Reads an argument "verbatim", between the following character and its next occurrence.
- (b) Not implemented! Only suitable in the argument specification of an environment, it denotes the body of the environment, between `\begin{`⟨*environment*⟩`}` and `\end{`⟨*environment*⟩`}`.

The types which define optional arguments are:

- o A standard LATEX optional argument, surrounded with square brackets, which will supply `nil` if not given (as described later).
- d Given as `d`⟨*token1*⟩⟨*token2*⟩, an optional argument which is delimited by ⟨*token1*⟩ and ⟨*token2*⟩. As with `o`, if no value is given `nil` is returned.
- O Given as `O{`⟨*default*⟩`}`, is like `o`, but returns ⟨*default*⟩ if no value is given.

D Given as `D`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`, it is as for `d`, but returns ⟨*default*⟩ if no value is given. Internally, the `o`, `d` and `O` types are short-cuts to an appropriated-constructed `D` type argument.

s An optional star, which will result in a value `true` if a star is present and `false` otherwise (as described later).

t An optional ⟨*token*⟩, which will result in a value `true` if ⟨*token*⟩ is present and `false` otherwise. Given as `t`⟨*token*⟩.

(e) Not implemented! Given as `e{`⟨*tokens*⟩`}`, a set of optional *embellishments*, each of which requires a *value*. If an embellishment is not present, `-NoValue-` is returned. Each embellishment gives one argument, ordered as for the list of ⟨*tokens*⟩ in the argument specification. All ⟨*tokens*⟩ must be distinct. *This is an experimental type.*

(E) Not implemented! As for `e` but returns one or more ⟨*defaults*⟩ if values are not given: `E{`⟨*tokens*⟩`}{`⟨*defaults*⟩`}`.

2.2 Function: `scan`

```
\input lparse.tex

\def\test{\par\directlua{
  local oarg, star, marg = lparse.scan('o s m')
  tex.print('o: ' .. tostring(oarg))
  tex.print('s: ' .. tostring(star))
  tex.print('m: ' .. tostring(marg))
}% Important: after \directlua no characters to expand
}

\test{marg} % o: nil s: false m: marg
\test[oarg]{marg} % o: oarg s: false m: marg
\test[oarg]*{marg} % o: oarg s: true m: marg

\bye
```

2.3 Function: `register_csname(csname, fn, opts)`

The function `register_csname(csname, fn, opts)` registers a Lua function under a control sequence name (csname). The first argument is the control sequence name without the leading backslash. The second argument is the Lua function to be called when the command name is called in the TeX code.

```
\input lparse.tex

\directlua{
  lparse.register_csname('lorem', function()
    tex.print('Lorem ipsum dolor')
  end)
}

\lorem % Lorem ipsum dolor

\bye
```

2.4 Auxiliary functions

Some auxiliary functions are exported in the `utils` table:

```
local lparse = require('lparse')

local parse_spec = lparse.utils.parse_spec
local scan_oarg = lparse.utils.scan_oarg
```

Function: `utils.scan_oarg(init_delim, end_delim)`

Plain TEX does not know optional arguments `[`⟨*oarg*⟩`]`. The function `scan_oarg` allows to search for optional arguments not only in LATEX but also in Plain TEX. The function uses the token library built into LuaTEX. The two parameters `init_delim` and `end_delim` can be omitted. Then square brackets are assumed to be delimiters. `utils.scan_oarg('(', ')')` searches for an optional argument in round brackets, for example. The function returns the string between the delimiters or `nil` if no delimiters could be found. The delimiters themselves are not included in the result. After the `\directlua{}`, the macro using `scan_oarg` must not expand to any characters.

```
\input lparse.tex

\def\test{\par\directlua{
  local oarg = lparse.utils.scan_oarg()
  tex.print('oarg: ' .. tostring(oarg))
}}

\test[oarg] % oarg: oarg
\test % oarg: nil

\bye
```

3 Known limitations

In some LuaLATEX environments, scanning for optional arguments (oargs) does not work. Unfortunately I don't know why. I would be very grateful for help.

3.1 In tabular environment

In a `tabular` environment, if a command that is scanned with `lparse` is preceded by any text, it does not work, otherwise it does.

```
\documentclass{article}
\begin{document}
\def\test{
  \directlua{
    local lparse = require('lparse')
    local oarg, marg = lparse.scan('O{} v')
    print(oarg, marg)
  }
}
\begin{tabular}{l}
\test{test} suffix \\
% No file test_error_tabular.aux.
%         test
```

```
%         test
% ! Missing } inserted.
% <inserted text>
%                 }
% l.17 \end
%           {tabular}
prefix \test{test} suffix \\
\end{tabular}
\end{document}
```

3.2 In multline environment

```
\documentclass{article}
\usepackage{amsmath}
\begin{document}
\def\test{
  \directlua{
    local lparse = require('lparse')
    local oarg, marg = lparse.scan('O{} v')
    print(oarg, marg)
  }
}
% 1        590
% 2        19
%         12
%
% ! Missing } inserted.
% <inserted text>
%                 }
% l.25 \end{multline*}
\begin{multline*}
p(x) = \test[1]{590}x^4y^2 + \test[2]{19}x^3y^3\\
- 12x^2y^4 - \test {12}xy^5
\end{multline*}
\end{document}
```

4 Implementation

The source code is hosted on Github. The following links will take you to the individual files:

- lparse.lua
- lparse.sty
- lparse.tex