# The lparse package

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

0.2.0 from 2025/06/19

```
\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 The `x` has been replaced by `l` because this package only works with LuaTEX.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 Class: `Scanner`

### 2.3.1 Method: `Scanner:scan()`

### 2.3.2 Method: `Scanner:export()`

### 2.3.3 Method: `Scanner:assert()`

### 2.3.4 Method: `Scanner:debug()`

## 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 Implementation

## 3.1 lparse.lua

```lua
-- lparse.lua
-- Copyright 2023-2025 Josef Friedrich
--
-- This work may be distributed and/or modified under the
-- conditions of the LaTeX Project Public License, either version 1.3c
-- of this license or (at your option) any later version.
-- The latest version of this license is in
--   http://www.latex-project.org/lppl.txt
-- and version 1.3c or later is part of all distributions of LaTeX
-- version 2008/05/04 or later.
--
-- This work has the LPPL maintenance status `maintained'.
--
-- The Current Maintainer of this work is Josef Friedrich.
--
-- This work consists of the files lparse.lua, lparse.tex,
-- and lparse.sty.
---
if lpeg == nil then
  lpeg = require('lpeg')
end

---
---@param spec string An argument specifier, for example `o m`
---
---Required arguments:
---
---* `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.
---
---Optional arguments:
---
---* `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 `token1`.
---  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`.
---
---@return Argument[]
local function parse_spec(spec)
  local V = lpeg.V
  local P = lpeg.P
  local Set = lpeg.S
  local Range = lpeg.R
  local CaptureFolding = lpeg.Cf
  local CaptureTable = lpeg.Ct
  local Cc = lpeg.Cc
  local CaptureSimple = lpeg.C

  local function add_result(result, value)
    if not result then
      result = {}
    end
    table.insert(result, value)
    return result
  end

  local function collect_delims(a, b)
    return { init_delim = a, end_delim = b }
  end

  local function collect_token(a)
    return { token = a }
  end

  local function set_default(a)
    return { default = a }
  end

  local function combine(...)
    local args = { ... }

    local output = {}

    for _, arg in ipairs(args) do
      if type(arg) ~= 'table' then
        arg = {}
      end

      for key, value in pairs(arg) do
        output[key] = value
      end

    end

    return output
  end

  local function ArgumentType(letter)
    local function get_type(l)
      return { argument_type = l }
    end
    return CaptureSimple(P(letter)) / get_type
  end

  local T = ArgumentType
```

```lua

  local pattern = P({
    'init',
    init = V('whitespace') ^ 0 *
      CaptureFolding(CaptureTable('') * V('list'), add_result),

    list = (V('arg') * V('whitespace') ^ 1) ^ 0 * V('arg') ^ -1,

    arg = V('m') + V('r') + V('R') + V('v') + V('o') + V('d') + V('O') +
      V('D') + V('s') + V('t'),

    m = T('m') / combine,

    r = T('r') * V('delimiters') / combine,

    R = T('R') * V('delimiters') * V('default') / combine,

    v = T('v') * Cc({ verbatim = true }) / combine,

    o = T('o') * Cc({ optional = true }) / combine,

    d = T('d') * V('delimiters') * Cc({ optional = true }) / combine,

    O = T('O') * V('default') * Cc({ optional = true }) / combine,

    D = T('D') * V('delimiters') * V('default') *
      Cc({ optional = true }) / combine,

    s = T('s') * Cc({ star = true }) / combine,

    t = T('t') * V('token') / combine,

    token = V('delimiter') / collect_token,

    delimiter = CaptureSimple(Range('!~')),

    delimiters = V('delimiter') * V('delimiter') / collect_delims,

    whitespace = Set(' \t\n\r'),

    default = P('{') * CaptureSimple((1 - P('}')) ^ 0) * P('}') /
      set_default,
  })

  return pattern:match(spec)

end

---
---@param t Token
local function debug_token(t)
  print(t)
  print('command', t.command)
  print('cmdname', t.cmdname)
  print('csname', t.csname)
  print('id', t.id)
  print('tok', t.tok)
  print('active', t.active)
  print('expandable', t.expandable)
  print('protected', t.protected)
  print('mode', t.mode)
  print('index', t.index)
end

```

```lua
---
---Scan for an optional delimited argument.
---
---@param init_delim? string # The character that marks the beginning of an optional argument (by default `[`).
---@param end_delim? string # The character that marks the end of an optional argument (by default `]`).
---
---@return string|nil # The string that was enclosed by the delimiters. The delimiters themselves are not returned.
local function scan_oarg(init_delim, end_delim)
  if init_delim == nil then
    init_delim = '['
  end
  if end_delim == nil then
    end_delim = ']'
  end

  ---
  ---@param t Token
  ---
  ---@return string
  local function convert_token_to_string(t)
    if t.index ~= nil then
      return utf8.char(t.index)
    else
      return '\\' .. t.csname
    end
  end

  local delimiter_stack = 0

  local function get_next_char()
    local t = token.get_next()
    local char = convert_token_to_string(t)
    if char == init_delim then
      delimiter_stack = delimiter_stack + 1
    end

    if char == end_delim then
      delimiter_stack = delimiter_stack - 1
    end
    return char, t
  end

  local char, t = get_next_char()

  if t.cmdname == 'spacer' then
    char, t = get_next_char()
  end

  if char == init_delim then
    local output = {}

    char, t = get_next_char()

    -- "while" better than "repeat ... until": The end_delimiter is
    -- included in the result output.
    while not (char == end_delim and delimiter_stack == 0) do
      table.insert(output, char)
      char, t = get_next_char()
    end
    return table.concat(output, '')
  else
```

```
250       token.put_next(t)
251     end
252   end
253
254   ---
255   ---Represents an argument of a command.
256   ---
257   ---The basic form of the argument specifier is a list of letters, where
258   ---each letter defines a `Argument`.
259   ---
260   ---## `m`:
261   ---
262   ---```lua
263   ---{ argument_type = 'm' }
264   ---```
265   ---
266   ---## `r`:
267   ---
268   ---```lua
269   ---{ argument_type = 'r', end_delim = '>', init_delim = '<' }
270   ---```
271   ---
272   ---## `R`:
273   ---
274   ---(`R<>{default}`)
275   ---
276   ---```lua
277   ---{
278   ---  argument_type = 'R',
279   ---  end_delim = '>',
280   ---  init_delim = '<',
281   ---  default = 'default',
282   ---}
283   ---```
284   ---
285   ---## `v`:
286   ---
287   ---```lua
288   ---{
289   ---  argument_type = 'v',
290   ---  verbatim = true,
291   ---}
292   ---```
293   ---
294   ---## `o`:
295   ---
296   ---```lua
297   ---{ argument_type = 'o', optional = true }
298   ---```
299   ---
300   ---## `d`:
301   ---
302   ---(`d<>`)
303   ---
304   ---```lua
305   ---{
306   ---  argument_type = 'd',
307   ---  optional = true,
308   ---  end_delim = '>',
309   ---  init_delim = '<',
310   ---}
311   ---```
312   ---
313   ---## `O`:
```

```
314   ---
315   ---(`O{default}`)
316   ---
317   ---```lua
318   ---{ argument_type = 'O', optional = true, default = 'default' }
319   ---```
320   ---
321   ---## `D`:
322   ---
323   ---(`D<>{default}`)
324   ---
325   ---```lua
326   ---{
327   ---  argument_type = 'D',
328   ---  optional = true,
329   ---  default = ' default ',
330   ---  end_delim = '>',
331   ---  init_delim = '<',
332   ---}
333   ---```
334   ---
335   ---
336   ---## `s`:
337   ---
338   ---```lua
339   ---{ argument_type = 's', star = true }
340   ---```
341   ---
342   ---
343   ---## `t`:
344   ---
345   ---```lua
346   ---{ argument_type = 't', token = '+' }
347   ---```
348   ---
349   ---@class Argument
350   ---@field argument_type? 'm' | 'r' | 'R' | 'v' | 'o' | 'd' | 'O' | 'D' | 's' | 't' A
      ↪  single letter representing the argument type in the list of letters.
351   ---@field optional? boolean Indicates whether the argument is optional.
352   ---@field init_delim? string The character that marks the beginning of an argument.
353   ---@field end_delim? string The character that marks the end of an argument.
354   ---@field star? boolean `true` if it is a star argument type (`s`).
355   ---@field default? string The default value if no value is given.
356   ---@field verbatim? boolean `true` if it is a verbatim argument type (`v`).
357   ---@field token? string The optional token for the argument type `t`.
358
359   ---A parser that parses the argument specification (list of letters).
360   ---@class Scanner
361   ---@field spec string An argument specifier
362   ---@field args Argument[]
363   ---@field result any[]
364   local Scanner = {}
365   ---@private
366   Scanner.__index = Scanner
367
368   ---
369   ---@param spec string An argument specifier, for example `o m`
370   ---
371   ---Required arguments:
372   ---
373   ---* `m`: A standard mandatory argument, which can either be a single
374   ---  token alone or multiple tokens surrounded by curly braces `{}`.
375   ---  Regardless of the input, the argument will be passed to the
376   ---  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.
---
---Optional arguments:
---
---* `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 `token1`.
---  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`.
function Scanner:new(spec)
  local parser = {}
  setmetatable(parser, Scanner)
  parser.spec = spec
  parser.args = parse_spec(spec)
  parser.result = parser:scan()
  return parser
end

---
---Scan for arguments in the token input stream.
---
---@return any[]
function Scanner:scan()
  local result = {}
  local index = 1
  for _, arg in pairs(self.args) do
    if arg.star then
      -- s
      result[index] = token.scan_keyword('*')
    elseif arg.token then
      -- t
      result[index] = token.scan_keyword(arg.token)
    elseif arg.optional then
      -- o d O D
      local oarg = scan_oarg(arg.init_delim, arg.end_delim)
      if arg.default and oarg == nil then
        oarg = arg.default
      end
      result[index] = oarg
```

```
    elseif arg.init_delim and arg.end_delim then
      -- r R
      local oarg = scan_oarg(arg.init_delim, arg.end_delim)
      if arg.default and oarg == nil then
        oarg = arg.default
      end
      if oarg == nil then
        tex.error('Missing required argument')
      end
      result[index] = oarg
    else
      -- m v
      local marg = token.scan_argument(arg.verbatim ~= true)
      if marg == nil then
        tex.error('Missing required argument')
      end
      result[index] = marg
    end
    index = index + 1
  end
  return result
end

---@private
function Scanner:set_result(...)
  self.result = { ... }
end

---
---@return string|boolean|nil ...
function Scanner:export()
  -- #self.arg: to get all elements of the result table, also elements
  -- with nil values.
  return table.unpack(self.result, 1, #self.args)
end

function Scanner:assert(...)
  local arguments = { ... }
  for index, arg in ipairs(arguments) do
    assert(self.result[index] == arg, string.format(
      'Argument at index %d doesn't match: "%s" != "%s"',
      index, self.result[index], arg))
  end
end

function Scanner:debug()
  for index = 1, #self.args do
    print(index, self.result[index])
  end
end

---
---@param spec string An argument specifier, for example `o m`
---
---Required arguments:
---
---* `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
```

```
505   ---  `token1` is missing, `nil` will be
506   ---  returned after a suitable error.
507   ---* `R` Given as `R` `token1` `token2` `default`,
508   ---  this is a required delimited argument as for `r`,
509   ---  but it has a user-definable recovery `default` instead of
510   ---  `nil`.
511   ---* `v`: Reads an argument `verbatim`, between the following
512   ---   character and its next occurrence.
513   ---
514   ---Optional arguments:
515   ---
516   ---* `o`: A standard LaTeX optional argument, surrounded with square
517   ---  brackets, which will supply
518   ---  `nil` if not given (as described later).
519   ---* `d`: Given as `d` `token1` `token2`, an optional
520   ---  argument which is delimited by `token1` and `token1`.
521   ---  As with  `o`, if no
522   ---  value is given `nil` is returned.
523   ---* `O`: Given as `O{default}`, is like `o`, but
524   ---  returns `default` if no value is given.
525   ---* `D`: Given as `D` `token1` `token2` `{default}`,
526   ---  it is as for `d`, but returns `default` if no value is given.
527   ---  Internally, the `o`, `d` and `O` types are
528   ---  short-cuts to an appropriated-constructed `D` type argument.
529   ---* `s`: An optional star, which will result in a value
530   ---  `true` if a star is present and `false`
531   ---  otherwise (as described later).
532   ---* `t`: An optional `token`, which will result in a value
533   ---  `true` if `token` is present and `false`
534   ---  otherwise. Given as `t` `token`.
535   ---
536   ---@return Scanner
537   local function create_scanner(spec)
538     return Scanner:new(spec)
539   end
540
541   ---
542   ---Scan for arguments in the token input stream.
543   ---
544   ---@param spec string An argument specifier, for example `o m`
545   ---
546   ---Required arguments:
547   ---
548   ---* `m`: A standard mandatory argument, which can either be a single
549   ---  token alone or multiple tokens surrounded by curly braces `{}`.
550   ---  Regardless of the input, the argument will be passed to the
551   ---  internal code without the outer braces. This is the `lparse`
552   ---  type specifier for a normal TeX argument.
553   ---* `r`: Given as `r` `token1` `token2`, this denotes a
554   ---  required delimited argument, where the delimiters are
555   ---  `token1` and `token2`. If the opening delimiter
556   ---  `token1` is missing, `nil` will be
557   ---  returned after a suitable error.
558   ---* `R` Given as `R` `token1` `token2` `default`,
559   ---  this is a required delimited argument as for `r`,
560   ---  but it has a user-definable recovery `default` instead of
561   ---  `nil`.
562   ---* `v`: Reads an argument `verbatim`, between the following
563   ---   character and its next occurrence.
564   ---
565   ---Optional arguments:
566   ---
567   ---* `o`: A standard LaTeX optional argument, surrounded with square
568   ---  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 `token1`.
---  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`.
---
---@return boolean|string|nil ...
local function scan(spec)
  local scanner = create_scanner(spec)
  return scanner:export()
end

return {
  scan = scan,
  Scanner = create_scanner,
  utils = { parse_spec = parse_spec, scan_oarg = scan_oarg },
}
```

## 3.2 lparse.tex

```
%% lparse.tex
%% Copyright 2023-2025 Josef Friedrich
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3c
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2008/05/04 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is Josef Friedrich.
%
% This work consists of the files lparse.lua, lparse.tex,
% and lparse.sty.

\directlua
{
  lparse = require('lparse')
}
```

## 3.3 lparse.sty

```
%% lparse.sty
%% Copyright 2023-2025 Josef Friedrich
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3c
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2008/05/04 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is Josef Friedrich.
%
% This work consists of the files lparse.lua, lparse.tex,
% and lparse.sty.

\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{lparse}[2025/06/19 v0.2.0 Parse and scan macro arguments in Lua on
↪  LuaTeX using a xparse like argument specification]

\input lparse.tex
```