The newcommand.py utility*

Scott Pakin
pakin@uiuc.edu

2001/05/29

Abstract

LaTeX's `\newcommand` is fairly limited in the way it processes optional arguments, but the Plain TeX alternative, a batch of `\def`s and `\futurelet`s, can be overwhelming to the casual LaTeX user. `newcommand.py` is a Python program that automatically generates LaTeX macro definitions for macros that require more powerful argument processing than `\newcommand` can handle. `newcommand.py` is intended for LaTeX advanced beginners (i.e., those who know how to use `\newcommand` but not internal LaTeX 2_ε commands like `\@ifnextchar`) and for more advanced users who want to save some typing when defining complex macros.

1 Introduction

LaTeX's `\newcommand` is a rather limited way to define new macros. Only one argument can be designated as optional, it must be the first argument, and it must appear within square brackets. Defining macros that take multiple optional arguments or in which an optional argument appears in the middle of the argument list is possible, but well beyond the capabilities of the casual LaTeX user. It requires using Plain TeX primitives such as `\def` and `\futurelet` and/or LaTeX 2_ε internal commands such as `\@ifnextchar`.

`newcommand.py` is a Python program that reads a specification of an argument list and automatically produces LaTeX code that processes the arguments appropriately. `newcommand.py` makes it easy to define LaTeX macros with more complex parameter parsing than is possible with `\newcommand` alone. Note that you do need to have Python installed on your system to run `newcommand.py`. If you don't, you can download it from `http://www.python.org`.

To define a LaTeX macro, one passes `newcommand.py` a macro description written in a simple specification language. The description essentially lists the required and optional arguments and, for each optional argument, the default value. The next section of this document describes the syntax and provides some examples,

*This file has version number 1.00a, last revised 2001/05/29.

but for now, let's look at how one would define the most trivial macro possible, one that takes no arguments. Enter the following at your operating system's prompt:

```
newcommand.py "MACRO trivial"
```

(Depending on your system, you may need to prefix that command with "`python`".) The program should output the following LaTeX code in response:

```
% Prototype: MACRO trivial
\newcommand{\trivial}{%
  % Put your code here.
}
```

Alternatively, you can run `newcommand.py` interactively, entering macro descriptions at the `% Prototype:` prompt:

```
% Prototype: MACRO trivial
\newcommand{\trivial}{%
  % Put your code here.
}
% Prototype:
```

Enter your operating system's end-of-file character (Ctrl-D in Unix or Ctrl-Z in Windows) to exit the program.

While you certainly don't need `newcommand.py` to write macros that are as trivial as `\trivial`, the previous discussion shows how to run the program and the sort of output that you should expect. There will always be a `Put your code here` comment indicating where you should fill in the actual macro code. At that location, all of the macro's parameters—both optional and required—will be defined and can be referred to in the ordinary way: `#1`, `#2`, `#3`, etc.

2 Usage

As we saw in the previous section, macros are defined by the word "`MACRO`" followed by the macro name, with no preceding backslash. Required arguments are entered `#1`, `#2`, `#3`, ..., with no surrounding braces:

```
PROMPT> newcommand.py "MACRO required #1 #2 #3 #4 #5"
% Prototype: MACRO required #1 #2 #3 #4 #5
\newcommand{\required}[5]{%
  % Put your code here.
}
```

("`PROMPT>`" represents your operating system's prompt; don't type it explicitly.)

Parameters must be numbered in a monotonically increasing order, starting with `#1` and going up to `#9`.[1] Incorrectly ordered parameters will produce an error message:

[1] TeX supports only nine macro arguments.

```
PROMPT> newcommand.py "MACRO required #1 #3 #4"
% Prototype: MACRO required #1 #3 #4
                                ^
newcommand.py: Saw parameter #3 when parameter #2 was expected.
```

Optional arguments are written as either "`OPT[`⟨*param*⟩`={`⟨*default*⟩`}]`" or "`OPT(`⟨*param*⟩`={`⟨*default*⟩`})`". In the former case, square brackets are used to offset the optional argument; in the latter case, parentheses are used. ⟨*param*⟩ is the parameter number (`#1`, `#2`, `#3`, ...), and ⟨*default*⟩ is the default value for that parameter. Note that the curly braces are required around ⟨*default*⟩.

```
PROMPT> newcommand.py "MACRO optional OPT[#1={maybe}]"
% Prototype: MACRO optional OPT[#1={maybe}]
\newcommand{\optional}[1][maybe]{%
  % Put your code here.
}
```

Up to this point, the examples have been so simple that `newcommand.py` is overkill for entering them. We can now begin specifying constructs that LaTeX's `\newcommand` can't handle, such as a parenthesized optional argument, an optional argument that doesn't appear at the beginning of the argument list, and multiple optional arguments:

```
PROMPT> newcommand.py
% Prototype: MACRO parenthesized OPT(#1={abc})
\makeatletter
\def\parenthesized{%
  \@ifnextchar({\parenthesized@i}{\parenthesized@i(abc)}%)
}
\def\parenthesized@i(#1){%
  % Put your code here.
}
\makeatother

% Prototype: MACRO nonbeginning #1 OPT[#2={abc}]
\makeatletter
\newcommand{\nonbeginning}[1]{%
  \@ifnextchar[{\nonbeginning@i#1}{\nonbeginning@i#1[abc]}%]
}
\def\nonbeginning@i#1[#2]{%
  % Put your code here.
}
\makeatother

% Prototype: MACRO multiple OPT[#1={abc}] OPT[#2={def}]
\makeatletter
\newcommand{\multiple}[1][abc]{%
  \@ifnextchar[{\multiple@ii[#1]}{\multiple@ii[#1][def]}%]
}
\def\multiple@ii[#1][#2]{%
```

```
  % Put your code here.
}
\makeatother

% Prototype:
```

In addition to required and optional parameters, it is also possible to specify text that must appear literally in the macro call. Merely specify it within curly braces:[2]

```
PROMPT> newcommand.py
% Prototype: MACRO textual #1 { and } #2 {.}
\def\textual#1 and #2.{%
  % Put your code here.
}
```

A macro such as `\textual` can be called like this:

```
\textual {Milk} and {cookies}.
```

Actually, in that example, because both `Milk` and `cookies` are delimited on the right by literal text, TEX can figure out how to split `\textual`'s argument into `#1` and `#2` even if the curly braces are omitted:

```
\textual Milk and cookies.
```

The template for optional arguments that was shown on the preceding page stated that optional arguments contain a "⟨*param*⟩`={`⟨*default*⟩`}`" specification. In fact, optional arguments can contain *multiple* "⟨*param*⟩`={`⟨*default*⟩`}`" specifications, as long as they are separated by literal text:

```
PROMPT> newcommand.py "MACRO multiopt OPT(#1={0},#2={0})"
% Prototype: MACRO multiopt OPT(#1={0},#2={0})
\makeatletter
\def\multiopt{%
  \@ifnextchar({\multiopt@i}{\multiopt@i(0,0)}%)
}
\def\multiopt@i(#1,#2){%
  % Put your code here.
}
\makeatother
```

In that example, `\multiopt` takes an optional parenthesized argument. If omitted, it defaults to `(0,0)`. If provided, the argument must be of the form "`(`⟨x⟩`,`⟨y⟩`)`". In either case, the comma-separated values within the parentheses are parsed into `#1` and `#2`. Contrast that with the following:

[2] Technically, the curly braces are required only if the argument contains a space, bracket, or parenthesis.

```
PROMPT> newcommand.py "MACRO multiopt OPT(#1={0,0})"
% Prototype: MACRO multiopt OPT(#1={0,0})
\makeatletter
\def\multiopt{%
  \@ifnextchar({\multiopt@i}{\multiopt@i(0,0)}%)
}
\def\multiopt@i(#1){%
  % Put your code here.
}
\makeatother
```

in which the optional argument still defaults to `(0,0)`, but `#1` receives *all* of the text that lies between the parentheses; `\multiopt` does not parse it into two comma-separated values in `#1` and `#2`, as it did in the previous example.

Summary

A macro is defined in `newcommand.py` with:

`MACRO` ⟨*name*⟩ ⟨*arguments*⟩

in which ⟨*name*⟩ is the name of the macro, and ⟨*arguments*⟩ is zero or more of the following:

Argument	Meaning	Example
`#`⟨*number*⟩	Parameter (required)	`#1`
`{`⟨*text*⟩`}`	Literal text (required)	`{+}`
`OPT[#`⟨*number*⟩`={`⟨*text*⟩`}]`	Parameter (optional, with default)	`OPT[#1={tbp}]`
`OPT(#`⟨*number*⟩`={`⟨*text*⟩`})`	Same as the above, but with parentheses instead of brackets	`OPT(#1={tbp})`

The braces surrounding literal text can be omitted if the text doesn't contain a space, bracket, or parenthesis. Within an `OPT` argument, `#`⟨*number*⟩`={`⟨*text*⟩`}` can be repeated any number of times, as long as the various instances are separated by literal text.

3 Further examples

The LaTeX `picture` environment takes two, parenthesized, coordinate-pair arguments, the second pair being optional. Here's how to define a macro that takes the same arguments as the `picture` environment and parses them into x_1, y_1, x_2, and y_2:

```
PROMPT> newcommand.py "MACRO picturemacro {(}#1,#2{)} OPT(#3={0},#4={0})"
% Prototype: MACRO picturemacro {(}#1,#2{)} OPT(#3={0},#4={0})
\makeatletter
\def\picturemacro(#1,#2){%
```

```
  \@ifnextchar({\picturemacro@i(#1,#2)}{\picturemacro@i(#1,#2)(0,0)}%)
}
\def\picturemacro@i(#1,#2)(#3,#4){%
  % Put your code here.
}
\makeatother
```

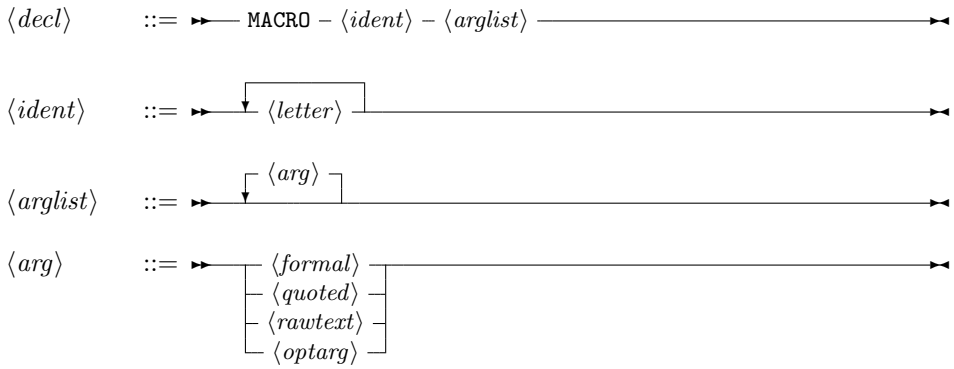
LaTeX's `\parbox` command takes three optional arguments and two required arguments. Furthermore, the third argument defaults to whatever value was specified for the first argument. This is easy to express in LaTeX with the help of `newcommand.py`:

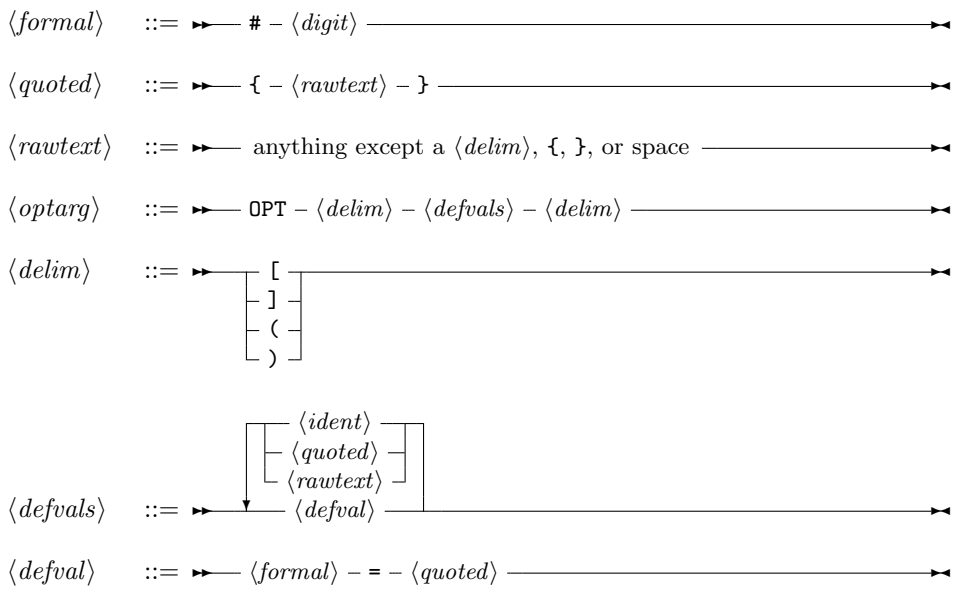
```
PROMPT> newcommand.py
% Prototype: MACRO parboxlike OPT[#1={s}] OPT[#2={\relax}] OPT[#3={#1}] #4 #5
\makeatletter
\newcommand{\parboxlike}[1][s]{%
  \@ifnextchar[{\parboxlike@ii[#1]}{\parboxlike@ii[#1][\relax]}%]
}
\def\parboxlike@ii[#1][#2]{%
  \@ifnextchar[{\parboxlike@iii[#1][#2]}{\parboxlike@iii[#1][#2][#1]}%]
}
\def\parboxlike@iii[#1][#2][#3]#4#5{%
  % Put your code here.
}
\makeatother

% Prototype:
```

4 Grammar

The following is the formal specification of `newcommand.py`'s grammar, written in a more-or-less top-down manner. Literal values, shown in a typewriter font, are case-sensitive. ⟨*letter*⟩ refers to a letter of the (English) alphabet. ⟨*digit*⟩ refers to a digit.





5 Future work

Two features I plan to add when I get around to it are support for defining LaTeX environments and support for starred versions of commands. The former should be fairly straightforward to do. The latter will take some thought to design; I'm not sure if the `*` should be considered an argument, which the function body can test with `\ifx`, or if it should cause one of two different functions to be called, depending on whether or not it's present. I'm leaning more towards considering it an argument, however. Send me e-mail if you think you'd actually use either of these features, and I'll boost its priority in my "to do" list.

I should probably also clean up the `newcommand.py` source code. It's my first Python program and the first parser I've written in a long, long time, so I'm sure it could be improved quite significantly.

6 Acknowledgements

`newcommand.py`'s parser uses John Aycock's Scanning, Parsing, and Rewriting Kit (SPARK). Thanks to him for writing this library and making it freely available and redistributable.