

The **primargs** package: Parsing arguments of primitives*

Bruno Le Floch

2012/08/04

Contents

1	primargs documentation	1
1.1	Read one token	2
1.2	Grabbing arguments	2
1.3	Discarding tokens	2
1.4	Afterassignment and <code>\globaldefs</code>	3
1.5	Internal functions	3
2	primargs implementation	3
2.1	Helpers	3
2.2	Read token with or without expansion	4
2.3	Removing tokens	5
2.3.1	One token	5
2.3.2	Optional space token	5
2.4	Right-hand sides of assignments	6
2.5	Get file name	8
3	Examples	10

1 primargs documentation

This L^AT_EX package is currently used by `morewrites` when redefining primitives: it allows to read arguments of primitives in place of T_EX. Of course, this is much slower than letting T_EX do things directly, but it may not be possible.

*This file has version number v0.2b, last revised 2012/08/04.

1.1 Read one token

<hr/> <hr/> <code>\g_primargs_token</code>	The result of <code>\primargs_read_x_token:N</code> and <code>\primargs_read_token:N</code> .
<hr/> <code>\primargs_read_x_token:N</code>	<code>\primargs_read_x_token:N</code> $\langle function \rangle$ Expands tokens recursively with <code>\exp_after:wN</code> , until encountering a non-expandable token, and calls the $\langle function \rangle$ afterwards. The non-expandable token found is stored as <code>\g_primargs_token</code> .
<hr/> <code>\primargs_read_token:N</code>	<code>\primargs_read_token:N</code> $\langle function \rangle$ Sets <code>\g_primargs_token</code> globally to the value of the next token, then calls the $\langle function \rangle$.

1.2 Grabbing arguments

A whole lot of functions have the same syntax: Finds what TeX's grammar calls a $\langle foo \rangle$ in the input stream, and feeds its value as a braced argument to the $\langle function \rangle$. Here is a list of the functions currently defined:

- `\primargs_get_number:N`
- `\primargs_get_dimen:N`
- `\primargs_get_glue:N`
- `\primargs_get_mudimen:N` (*not implemented properly*)
- `\primargs_get_muglue:N`
- `\primargs_get_general_text:N`
- `\primargs_get_file_name:N`

1.3 Discarding tokens

<hr/> <hr/> <code>\primargs_remove_token:N</code>	<code>\primargs_remove_token:N</code> $\langle function \rangle$ Removes one $\langle token \rangle$ following the $\langle function \rangle$, which is performed next.
<hr/> <code>\primargs_remove_one_optional_space:N</code>	<code>\primargs_remove_one_optional_space:N</code> $\langle function \rangle$ Removes $\langle one\ optional\ space \rangle$ after the $\langle function \rangle$.
<hr/> <code>\primargs_remove_optional_spaces:N</code>	<code>\primargs_remove_optional_spaces:N</code> $\langle function \rangle$ Removes $\langle optional\ spaces \rangle$ after the $\langle function \rangle$.
<hr/> <code>\primargs_remove_equals:N</code>	<code>\primargs_remove_equals:N</code> $\langle function \rangle$ Removes $\langle equals \rangle$, namely $\langle optional\ spaces \rangle$ followed optionally by an explicit = character with category other.

1.4 Afterassignment and \globaldefs

The `\globaldefs` parameter is not taken into account yet, and setting it to a non-zero value may make everything crash. It is straightforward to fix that for negative `\globaldefs`, but positive `\globaldefs` make things complicated.

Tokens inserted using `\afterassignment` may be lost when using this package, since it uses `\afterassignment` internally.

1.5 Internal functions

`\@@_get_rhs:NnN`
`\@@_get_rhs:NoN`

`\@@_get_rhs:NnN <register> {<register rhs>} <function>`

Use the `<register>` to find a right-hand side of a valid assignment for this type of variable, and feed the value found to the `<function>`. The value of the `<register>` is then restored using `<register> = <register rhs>`, where the `<register rhs>` should be the initial value of the `<register>`. All those assignments are performed within a group, but some are automatically global, and `\globaldefs` may cause trouble with others.

2 primargs implementation

`<*package>`

```

1 \RequirePackage{expl3}
2 \ProvidesExplPackage
3   {primargs}{2012/08/04}{0.2b}{Parsing arguments of primitives}
4 <@@=primargs>
```

2.1 Helpers

`\g__primargs_code_tl`

Used to contain temporary code.

```

5 \cs_new_protected_nopar:Npn \g__primargs_code_tl { }
(End definition for \g__primargs_code_tl)
```

`_primargs_no_afterassignment:`

Supersedes any `\afterassignment` token.

```

6 \cs_new_protected_nopar:Npn \_primargs_no_afterassignment:
7   { \tex_afterassignment:D \tex_relax:D }
(End definition for \_primargs_no_afterassignment:)
```

`_primargs_no_localdefs:`

This function, which must be called in a group, ensures that `\global` indeed makes definitions global.

```

8 \cs_new_protected_nopar:Npn \_primargs_no_localdefs:
9   {
10     \_primargs_no_afterassignment:
11     \int_compare:nNf \tex_globaldefs:D > \c_zero
12       { \tex_globaldefs:D = \c_zero }
13   }
(End definition for \_primargs_no_localdefs:)
```

2.2 Read token with or without expansion

TeX often calls the `get_x_token` procedure when parsing various parts of its grammar. This expands tokens recursively until reaching a non-expandable token. We emulate this by expanding with `\expandafter`, then checking whether the upcoming token is expandable or not using `\futurelet`, and if it is, expanding again.

One thing to be careful about is that

```
\expandafter \show \noexpand \space
```

shows the \meaning of the \notexpanded: \space, namely \relax (frozen, in fact, hence a bit different from the normal \relax), while expanding twice with

```
\expandafter \expandafter \expandafter \show \noexpand \space
```

expands the \space to the underlying space character token. What this means is that we must first check if the token is expandable or not, and only then expand, and that the token should not be queried again using `\futurelet`. On this latter point, run

```
\def \test { \show \next \futurelet \next \test }
\expandafter \test \noexpand \space
```

to see how `\next` changes from `\relax` to becoming a macro.

```
\primargs_read_x_token:N
\__primargs_read_x_token:N
  \__primargs_read_x_token_ii:N
```

First query the following token. Then test whether it is expandable, using a variant of the `\token_if_expandable:N` test.¹ If the token is expandable, `\exp_not:N` will change its \meaning to `\relax`, the test is false, we expand, and call the loop. Otherwise, we stop. Interestingly, we don't ever need to take the user's function as an argument.

```
14 \cs_new_protected_nopar:Npn \primargs_read_x_token:N
15   {
16     \group_begin:
17       \__primargs_no_localdefs:
18       \__primargs_read_x_token:N
19   }
20 \cs_new_protected_nopar:Npn \__primargs_read_x_token:N
21   {
22     \tex_afterassignment:D \__primargs_read_x_token_ii:N
23     \tex_global:D \tex_futurelet:D \g_primargs_token
24   }
25 \cs_new_protected_nopar:Npn \__primargs_read_x_token_ii:N
26   {
27     \exp_after:wN
28     \if_meaning:w \exp_not:N \g_primargs_token \g_primargs_token
29     \group_end:
30     \exp_after:wN \use_none:nnn
31     \fi:
32     \exp_after:wN \__primargs_read_x_token:N \exp_after:wN
33   }
```

¹This L^AT_EX3 test returns false for undefined tokens (by design), but TeX's `get_x_token` expands those undefined tokens, causing errors, so we should as well.

(End definition for `\primargs_read_x_token:N` This function is documented on page 2.)

`\primargs_read_token:N` The same without expansion, for use when we already know that what follows is expanded.

```

34 \cs_new_protected_nopar:Npn \primargs_read_token:N
35   {
36     \group_begin:
37       \__primargs_no_localdefs:
38       \tex_afterassignment:D \group_end:
39       \tex_global:D \tex_futurelet:D \g_primargs_token
40   }

```

(End definition for `\primargs_read_token:N` This function is documented on page 2.)

2.3 Removing tokens

2.3.1 One token

`\primargs_remove_token:N` Remove token using `\let` (note the presence of = and a space, to correctly remove explicit space characters), then insert the `\function` with `\afterassignment`.

```

41 \cs_new_protected:Npn \primargs_remove_token:N #1
42   {
43     \group_begin:
44       \__primargs_no_localdefs:
45       \tex_aftergroup:D #1
46       \tex_afterassignment:D \group_end:
47       \tex_global:D \tex_let:D \g_primargs_token = ~
48   }

```

(End definition for `\primargs_remove_token:N` This function is documented on page 2.)

2.3.2 Optional space token

`\primargs_remove_one_optional_space:N` Start a group: we will insert the `\function` at its end.

```

\__primargs_remove_one_optional_space:
49 \cs_new_protected:Npn \primargs_remove_one_optional_space:N #1
50   {
51     \group_begin:
52       \__primargs_no_localdefs:
53       \tex_aftergroup:D #1
54       \primargs_read_x_token:N \__primargs_remove_one_optional_space:
55   }
56 \exp_args:NNo \cs_new_protected_nopar:Npn \__primargs_remove_one_optional_space:
57   {
58     \use:n { \if_catcode:w } ~ \exp_not:N \g_primargs_token
59     \exp_after:wN \primargs_remove_token:N
60     \fi:
61     \group_end:
62   }

```

(End definition for `\primargs_remove_one_optional_space:N` This function is documented on page 2.)

`\primargs_remove_optional_spaces:N` Start a group: we will insert the $\langle function \rangle$ at its end.

```

\__primargs_remove_optional_spaces:
\__primargs_remove_optional_spaces_ii:
63 \cs_new_protected:Npn \primargs_remove_optional_spaces:N #1
64 {
65   \group_begin:
66   \__primargs_no_localdefs:
67   \tex_aftergroup:D #1
68   \__primargs_remove_optional_spaces:
69 }
70 \cs_new_protected_nopar:Npn \__primargs_remove_optional_spaces:
71 { \primargs_read_x_token:N \__primargs_remove_optional_spaces_ii: }
72 \exp_args:NNo \cs_new_protected_nopar:Npn \__primargs_remove_optional_spaces_ii:
73 {
74   \use:n { \if_catcode:w } ~ \exp_not:N \g_primargs_token
75   \exp_after:wN \primargs_remove_token:N
76   \exp_after:wN \__primargs_remove_optional_spaces:
77   \else:
78   \exp_after:wN \group_end:
79   \fi:
80 }

```

(End definition for `\primargs_remove_optional_spaces:N` This function is documented on page 2.)

`\primargs_remove_equals:N` Remove $\langle optional spaces \rangle$, then test for an explicit `=`, both in `\meaning` and as a token list (once we know its `\meaning`, we can grab it safely).

```

81 \cs_new_protected:Npn \primargs_remove_equals:N #1
82 {
83   \group_begin:
84   \tex_aftergroup:D #1
85   \primargs_remove_optional_spaces:N \__primargs_remove_equals:
86 }
87 \cs_new_protected_nopar:Npn \__primargs_remove_equals:
88 {
89   \if_meaning:w = \g_primargs_token
90   \exp_after:wN \__primargs_remove_equals_ii:NN
91   \fi:
92   \group_end:
93 }
94 \cs_new_protected:Npn \__primargs_remove_equals_ii:NN #1#2
95 { \tl_if_eq:nnTF { #2 } { = } { #1 } { #1 #2 } }

```

(End definition for `\primargs_remove_equals:N` This function is documented on page 2.)

2.4 Right-hand sides of assignments

The naive approach to reading an integer, or a general text, is to let \TeX perform an assignment to a `\count`, or a `\toks`, register and regain control using `\afterassignment`. The question is then to know which `\count` or `\toks` register to use. One might think that any can be used as long as the assignment happens in a group.

However, there comes the question of the `\globaldefs` parameter. If this parameter is positive, every assignment is global, including assignments to the parameter itself,

preventing us from setting it to zero locally; hence, we are stuck with global assignments (if `\globaldefs` is negative, we can change it, locally, to whatever value pleases us, as done by `__primargs_no_localdefs:`). We may thus not use scratch registers to parse integers, general texts, and other pieces of T_EX's grammar.

For integers, we will use `\deadcycles`, a parameter which is automatically assigned globally, and we revert it to its previous value afterwards. For general text, we will use `\errhelp`, which we will assign locally if possible (`\globaldefs` negative or zero), and otherwise reset to be empty.

`__primargs_get_rhs:NnN` The last two lines of this function are the key: assign to `#1`, then take control using `\afterassignment`. After the assignment, we expand the value found, `\tex_the:D #1`, within a brace group, then restore `#1` using its initial value `#2`, and end the group. The earlier use of `\tex_aftergroup:D` inserts the *function* `#3` before the brace group containing the value found.

```

96 \cs_new_protected:Npn \__primargs_get_rhs:NnN #1#2#3
97   {
98     \group_begin:
99     \__primargs_no_localdefs:
100     \tex_aftergroup:D #3
101     \tl_gset:Nn \g__primargs_code_tl
102       {
103         \use:x
104         {
105           \exp_not:n { #1 = #2 \group_end: }
106           { \tex_the:D #1 }
107         }
108       }
109     \tex_afterassignment:D \g__primargs_code_tl
110     #1 =
111   }
112 \cs_generate_variant:Nn \__primargs_get_rhs:NnN { No }

```

(End definition for __primargs_get_rhs:NnN and __primargs_get_rhs:NoN)

`\primargs_get_number:N` We use the general `__primargs_get_rhs:NoN`, using the internal register `\deadcycles`, for which all assignments are global: thus, restoring its value will not interact badly with groups.

```

113 \cs_new_protected_nopar:Npn \primargs_get_number:N
114   {
115     \__primargs_get_rhs:NoN \tex_deadcycles:D
116     { \tex_the:D \tex_deadcycles:D }
117   }

```

(End definition for \primargs_get_number:N This function is documented on page ??.)

`\primargs_get_dimen:N` Use `\hoffset` as a register since it is not too likely to be changed locally (anyways, which register we use is not that important since normally, `\globaldefs` is zero, and everything is done within a group).

```

118 \cs_new_protected_nopar:Npn \primargs_get_dimen:N
119   {

```

```

120     \__primargs_get_rhs:NoN \tex_hoffset:D
121     { \tex_the:D \tex_hoffset:D }
122   }
(End definition for \primargs_get_dimen:N This function is documented on page ??.)

```

```

\primargs_get_glue:N Use \topskip.
123 \cs_new_protected_nopar:Npn \primargs_get_glue:N
124   {
125     \__primargs_get_rhs:NoN \tex_topskip:D
126     { \tex_the:D \tex_topskip:D }
127   }
(End definition for \primargs_get_glue:N This function is documented on page ??.)

```

```

\primargs_get_mudimen:N There is no such thing as a ⟨mudimen variable⟩, so we're on our own to parse a ⟨mudimen⟩.
Warn about that problem, and parse a ⟨muglue⟩ instead.
128 \cs_new_protected_nopar:Npn \primargs_get_mudimen:N
129   {
130     \msg_warning:nn { primargs } { get-mudimen }
131     \primargs_get_muglue:N
132   }
133 \msg_new:nnn { primargs } { get-mudimen }
134   { The~\iow_char:N\primargs_get_mudimen:N-function-is~buggy. }
(End definition for \primargs_get_mudimen:N This function is documented on page ??.)

```

```

\primargs_get_muglue:N Use \thinmuskip.
135 \cs_new_protected_nopar:Npn \primargs_get_muglue:N
136   {
137     \__primargs_get_rhs:NoN \tex_thinmuskip:D
138     { \tex_the:D \tex_thinmuskip:D }
139   }
(End definition for \primargs_get_muglue:N This function is documented on page ??.)

```

```

\primargs_get_general_text:N For a general text, use \errhelp, since it shouldn't be a big problem if that's changed.
We don't revert it to its value, but to be empty (note the extra braces, though, since
it's a token register), because it is probably better to have no help than the wrong help
hanging around. Besides, \errhelp is always set for immediate use.
140 \cs_new_protected_nopar:Npn \primargs_get_general_text:N
141   { \__primargs_get_rhs:NoN \tex_errhelp:D { { } } }
(End definition for \primargs_get_general_text:N This function is documented on page ??.)

```

2.5 Get file name

```

\g__primargs_file_name_tl Token list used to build a file name, one character at a time.
142 \tl_new:N \g__primargs_file_name_tl
(End definition for \g__primargs_file_name_tl This variable is documented on page ??.)

```


`\primargs_get_file_name:N` Empty the file name (globally), and build it one character at a time. The *⟨function⟩* is added at the end of a group, started here. As described in the *TeXbook*, a *⟨file name⟩* should start with *⟨optional spaces⟩*, which we remove, then character tokens, ending with a non-expandable character or control sequence. After space removal, `\g_primargs_token` contains the next token, so no need for `\primargs_read_token:N`.

```

143 \cs_new_protected:Npn \primargs_get_file_name:N #1
144 {
145   \group_begin:
146   \__primargs_no_localdefs:
147   \tex_aftergroup:D #1
148   \tl_gclear:N \g__primargs_file_name_tl
149   \primargs_remove_optional_spaces:N \__primargs_get_file_name_test:
150 }

```

(End definition for `\primargs_get_file_name:N` This function is documented on page ??.)

`__primargs_get_file_name_test:` The token read is in `\g_primargs_token`, and is non-expandable. If it is a control sequence, end the *⟨file name⟩*. If it is a space, the the *⟨file name⟩* ends after its removal. Otherwise, we extract the character from the *\meaning* of the *⟨token⟩*, which we remove anyways: in that case, we'll recurse.

```

151 \cs_new_protected_nopar:Npn \__primargs_get_file_name_test:
152 {
153   \token_if_cs:NTF \g_primargs_token
154   { \__primargs_get_file_name_end: }
155   {
156     \token_if_eq_charcode:NNTF \c_space_token \g_primargs_token
157     { \primargs_remove_token:N \__primargs_get_file_name_end: }
158     { \primargs_remove_token:N \__primargs_get_file_name_char: }
159   }
160 }

```

(End definition for `__primargs_get_file_name_test:`)

`__primargs_get_file_name_end:` When the end of the file name is reached, end the group, after expanding the contents of `\g__primargs_file_name_tl`.

```

161 \cs_new_protected_nopar:Npn \__primargs_get_file_name_end:
162 { \exp_args:No \group_end: \g__primargs_file_name_tl }

```

(End definition for `__primargs_get_file_name_end:`)

`__primargs_get_file_name_char:` With an explicit character, applying `\string` would give the character code. Here, implicit characters have to be converted too, so we must work with the *\meaning*, which is two or three words separated by spaces, then the character. The *ii* auxiliary removes the first two words, and duplicates the remainder (either one character, or a word and a character), and the second auxiliary leaves the second piece in the definition (in both cases, the character). Then loop with expansion.

```

163 \cs_new_protected_nopar:Npn \__primargs_get_file_name_char:
164 {
165   \tl_gput_right:Nx \g__primargs_file_name_tl
166   {

```

```

167         \exp_after:wN \__primargs_get_file_name_char_ii:w
168         \token_to_meaning:N \g_primargs_token
169         \q_stop
170     }
171     \primargs_read_x_token:N \__primargs_get_file_name_test:
172 }
173 \cs_new:Npn \__primargs_get_file_name_char_ii:w #1 ~ #2 ~ #3 \q_stop
174 { \__primargs_get_file_name_char_iii:w #3 ~ #3 ~ \q_stop }
175 \cs_new:Npn \__primargs_get_file_name_char_iii:w #1 ~ #2 ~ #3 \q_stop {#2}
(End definition for \__primargs_get_file_name_char: This function is documented on page ??.)

```

3 Examples

[Old text which I don't want to remove yet.] A few examples of what this package can parse.

- $\langle integer \rangle$ denotes an integer in any form that \TeX accepts as the right-hand side of a primitive integer assignment of the form $\text{\count0}=\langle integer \rangle$;
- $\langle equals \rangle$ is an arbitrary (optional) number of explicit or implicit space characters, an optional explicit equal sign of category other, and further (optional) explicit or implicit space characters;
- $\langle file\ name \rangle$ is an arbitrary sequence of explicit or implicit characters with arbitrary category codes (except active characters, which are expanded before reaching \TeX 's mouth), ending either with a space character (character code 32, arbitrary non-active category code, explicit or implicit), which is removed, or with a non-expandable token, with some care needed for the case of a \notexpanded : expandable token;
- $\langle filler \rangle$ is an arbitrary combination of tokens each of whose meaning is \relax or a character with category code 10;
- $\langle general\ text \rangle$ is a $\langle filler \rangle$, followed by braced tokens, starting with an explicit or implicit begin-group character, and ending with the matching explicit end-group character (both with any character code), with an equal number of explicit begin-group and end-group characters in between: this is precisely the right-hand side of an assignment of the form $\text{\toks0}=\langle general\ text \rangle$.

$\langle /package \rangle$