The primargs package: Parsing arguments of primitives*

Bruno Le Floch

2013/01/08

Contents

1 primargs documentation

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

*This file has version number v0.2e, last revised 2013/01/08.

1.1 Read one token

`\g_primargs_token` The result of `\primargs_read_x_token:N` and `\primargs_read_token:N`.

`\primargs_read_x_token:N` `\primargs_read_x_token:N` ⟨*function*⟩

Expands tokens recursively with `\exp_after:wN`, until encountering a non-expandable token, and calls the ⟨*function*⟩ afterwards. The non-expandable token found is stored as `\g_primargs_token`.

`\primargs_read_token:N` `\primargs_read_token:N` ⟨*function*⟩

Sets `\g_primargs_token` globally to the value of the next token, then calls the ⟨*function*⟩.

1.2 Grabbing arguments

`\primargs_get_number:N` `\primargs_get_dimen:N` `\primargs_get_glue:N` `\primargs_get_mudimen:N` `\primargs_get_muglue:N` `\primargs_get_general_text:N` `\primargs_get_file_name:N` `\primargs_get_`⟨*foo*⟩`:N` ⟨*function*⟩

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

1.3 Discarding tokens

`\primargs_remove_token:N` `\primargs_remove_token:N` ⟨*function*⟩

Removes one ⟨*token*⟩ following the ⟨*function*⟩, which is performed next.

`\primargs_remove_one_optional_space:N` `\primargs_remove_one_optional_space:N` ⟨*function*⟩

Removes ⟨*one optional space*⟩ after the ⟨*function*⟩.

`\primargs_remove_optional_spaces:N` `\primargs_remove_optional_spaces:N` ⟨*function*⟩

Removes ⟨*optional spaces*⟩ after the ⟨*function*⟩.

`\primargs_remove_equals:N` `\primargs_remove_equals:N` ⟨*function*⟩

Removes ⟨*equals*⟩, namely ⟨*optional spaces*⟩ 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>

```
\RequirePackage {expl3} [2012/08/14]
\ProvidesExplPackage
  {primargs} {2013/01/08} {0.2e} {Parsing arguments of primitives}
```

⟨@@=primargs⟩

2.1 Helpers

\g__primargs_code_tl

Used to contain temporary code.

```
\cs_new_protected_nopar:Npn \g__primargs_code_tl { }
```

(*End definition for* \g__primargs_code_tl.)

__primargs_no_afterassignment:

Supersedes any \afterassignment token.

```
\cs_new_protected_nopar:Npn \__primargs_no_afterassignment:
  { \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.

```
\cs_new_protected_nopar:Npn \__primargs_no_localdefs:
  {
    \__primargs_no_afterassignment:
    \int_compare:nNnF \tex_globaldefs:D > \c_zero
      { \tex_globaldefs:D = \c_zero }
  }
```

(*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:NTF` 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.

```
\cs_new_protected_nopar:Npn \primargs_read_x_token:N
  {
    \group_begin:
      \__primargs_no_localdefs:
      \__primargs_read_x_token:N
  }
\cs_new_protected_nopar:Npn \__primargs_read_x_token:N
  {
    \tex_afterassignment:D \__primargs_read_x_token_ii:N
    \tex_global:D \tex_futurelet:D \g_primargs_token
  }
\cs_new_protected_nopar:Npn \__primargs_read_x_token_ii:N
  {
    \exp_after:wN
    \if_meaning:w \exp_not:N \g_primargs_token \g_primargs_token
      \group_end:
      \exp_after:wN \use_none:nnn
    \fi:
    \exp_after:wN \__primargs_read_x_token:N \exp_after:wN
  }
```

[1] This LaTeX3 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.

```
\cs_new_protected_nopar:Npn \primargs_read_token:N
  {
    \group_begin:
      \__primargs_no_localdefs:
      \tex_afterassignment:D \group_end:
      \tex_global:D \tex_futurelet:D \g_primargs_token
  }
```

(*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`.

```
\cs_new_protected:Npn \primargs_remove_token:N #1
  {
    \group_begin:
      \__primargs_no_localdefs:
      \tex_aftergroup:D #1
      \tex_afterassignment:D \group_end:
      \tex_global:D \tex_let:D \g_primargs_token = ~
  }
```

(*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:`

```
\cs_new_protected:Npn \primargs_remove_one_optional_space:N #1
  {
    \group_begin:
      \__primargs_no_localdefs:
      \tex_aftergroup:D #1
      \primargs_read_x_token:N \__primargs_remove_one_optional_space:
  }
\exp_args:NNo \cs_new_protected_nopar:Npn \__primargs_remove_one_optional_space:
  {
    \use:n { \if_catcode:w } ~ \exp_not:N \g_primargs_token
      \exp_after:wN \primargs_remove_token:N
    \fi:
    \group_end:
  }
```

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

\primargs_remove_optional_spaces:N
__primargs_remove_optional_spaces:
__primargs_remove_optional_spaces_ii:

Start a group: we will insert the ⟨*function*⟩ at its end.

```
\cs_new_protected:Npn \primargs_remove_optional_spaces:N #1
  {
    \group_begin:
      \__primargs_no_localdefs:
      \tex_aftergroup:D #1
      \__primargs_remove_optional_spaces:
  }
\cs_new_protected_nopar:Npn \__primargs_remove_optional_spaces:
  { \primargs_read_x_token:N \__primargs_remove_optional_spaces_ii: }
\exp_args:NNo \cs_new_protected_nopar:Npn \__primargs_remove_optional_spaces_ii:
  {
    \use:n { \if_catcode:w } ~ \exp_not:N \g_primargs_token
      \exp_after:wN \primargs_remove_token:N
      \exp_after:wN \__primargs_remove_optional_spaces:
    \else:
      \exp_after:wN \group_end:
    \fi:
  }
```

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

\primargs_remove_equals:N

Remove ⟨*optional spaces*⟩, then test for an explicit `=`, both in `\meaning` and as a token list (once we know its `\meaning`, we can grab it safely).

```
\cs_new_protected:Npn \primargs_remove_equals:N #1
  {
    \group_begin:
      \tex_aftergroup:D #1
      \primargs_remove_optional_spaces:N \__primargs_remove_equals:
  }
\cs_new_protected_nopar:Npn \__primargs_remove_equals:
  {
      \if_meaning:w = \g_primargs_token
        \exp_after:wN \__primargs_remove_equals_ii:NN
      \fi:
    \group_end:
  }
\cs_new_protected:Npn \__primargs_remove_equals_ii:NN #1#2
  { \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 TEX'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`
`\__primargs_get_rhs:NoN`

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.

```
\cs_new_protected:Npn \__primargs_get_rhs:NnN #1#2#3
  {
    \group_begin:
      \__primargs_no_localdefs:
      \tex_aftergroup:D #3
      \tl_gset:Nn \g__primargs_code_tl
        {
          \use:x
            {
              \exp_not:n { #1 = #2 \group_end: }
              { \tex_the:D #1 }
            }
        }
      \tex_afterassignment:D \g__primargs_code_tl
      #1 =
  }
\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.

```
\cs_new_protected_nopar:Npn \primargs_get_number:N
  {
    \__primargs_get_rhs:NoN \tex_deadcycles:D
      { \tex_the:D \tex_deadcycles:D }
  }
```

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

`\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).

```
\cs_new_protected_nopar:Npn \primargs_get_dimen:N
  {
```

```
    \__primargs_get_rhs:NoN \tex_hoffset:D
      { \tex_the:D \tex_hoffset:D }
  }
```

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

`\primargs_get_glue:N` Use `\topskip`.

```
\cs_new_protected_nopar:Npn \primargs_get_glue:N
  {
    \__primargs_get_rhs:NoN \tex_topskip:D
      { \tex_the:D \tex_topskip:D }
  }
```

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

`\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.

```
\cs_new_protected_nopar:Npn \primargs_get_mudimen:N
  {
    \msg_warning:nn { primargs } { get-mudimen }
    \primargs_get_muglue:N
  }
\msg_new:nnn { primargs } { get-mudimen }
  { The~\iow_char:N\\primargs_get_mudimen:N~function~is~buggy. }
```

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

`\primargs_get_muglue:N` Use `\thinmuskip`.

```
\cs_new_protected_nopar:Npn \primargs_get_muglue:N
  {
    \__primargs_get_rhs:NoN \tex_thinmuskip:D
      { \tex_the:D \tex_thinmuskip:D }
  }
```

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

`\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.

```
\cs_new_protected_nopar:Npn \primargs_get_general_text:N
  { \__primargs_get_rhs:NoN \tex_errhelp:D { { } } }
```

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

2.5 Get file name

`\g__primargs_file_name_tl` Token list used to build a file name, one character at a time.

```
\tl_new:N \g__primargs_file_name_tl
```

(*End definition for* `\g__primargs_file_name_tl`.)

\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`.

```
\cs_new_protected:Npn \primargs_get_file_name:N #1
  {
    \group_begin:
      \__primargs_no_localdefs:
      \tex_aftergroup:D #1
      \tl_gclear:N \g__primargs_file_name_tl
      \primargs_remove_optional_spaces:N \__primargs_get_file_name_test:
  }
```

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

__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.

```
\cs_new_protected_nopar:Npn \__primargs_get_file_name_test:
  {
    \token_if_cs:NTF \g_primargs_token
      { \__primargs_get_file_name_end: }
      {
        \token_if_eq_charcode:NNTF \c_space_token \g_primargs_token
          { \primargs_remove_token:N \__primargs_get_file_name_end: }
          { \primargs_remove_token:N \__primargs_get_file_name_char: }
      }
  }
```

(*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`.

```
\cs_new_protected_nopar:Npn \__primargs_get_file_name_end:
  { \exp_args:No \group_end: \g__primargs_file_name_tl }
```

(*End definition for* `\__primargs_get_file_name_end:`.)

__primargs_get_file_name_char:
__primargs_get_file_name_char_ii:w
__primargs_get_file_name_char_iii:w

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. This technique would fail if the character could be a space (character code 32).

```
\cs_new_protected_nopar:Npn \__primargs_get_file_name_char:
  {
    \tl_gput_right:Nx \g__primargs_file_name_tl
```

```
      {
        \exp_after:wN \__primargs_get_file_name_char_ii:w
        \token_to_meaning:N \g_primargs_token
        \q_stop
      }
    \primargs_read_x_token:N \__primargs_get_file_name_test:
  }
\cs_new:Npn \__primargs_get_file_name_char_ii:w #1 ~ #2 ~ #3 \q_stop
  { \__primargs_get_file_name_char_iii:w #3 ~ #3 ~ \q_stop }
\cs_new:Npn \__primargs_get_file_name_char_iii:w #1 ~ #2 ~ #3 \q_stop {#2}
```

(*End definition for* `\__primargs_get_file_name_char:`.)

3 Examples

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

- ⟨*integer*⟩ denotes an integer in any form that TEX accepts as the right-hand side of a primitive integer assignment of the form `\count0=`⟨*integer*⟩;
- ⟨*equals*⟩ 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;
- ⟨*file name*⟩ 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;
- ⟨*filler*⟩ is an arbitrary combination of tokens each of whose meaning is `\relax` or a character with category code 10;
- ⟨*general text*⟩ is a ⟨*filler*⟩, 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 `\toks0=`⟨*general text*⟩.

</package>