Consolidation of expl3

Morten Høgholm

LaTeX Project

TUG 2009, Notre Dame University

Outline

A mixture

- TeX is both macro programming and document level language
- plain TeX and LaTeX provide a solution: `@` is used to signal internal command.
- However, many internal commands do not use `@`, e.g., all primitives.
- Many good names taken: `\box`, `\special`, etc.

Typical problems

This is the source of some typical communications with TEX.

- `Use of \@ doesn't match its definition`
- `You can't use \spacefactor in vertical (or math) mode.`
- Spurious spaces.
- % is a very common symbol when doing definitions.

All because TEX and LATEX have no proper low-level API.

A real API

Measures have been taken to improve the situation

- A programming environment where all white space is ignored
- A consistent naming scheme using module name, description and possibly data type.
- `_` used to enhance readability of names: `\c_module_magic_int`
- Colon used in function names to signal argument signature: `\foo_bar:nn` is a function taking two arguments.
- This is turned on and off with `\ExplSyntaxOn` and `\ExplSyntaxOff`
- Just load the package expl3

An example

```
\seq_new:N \g_mho_example_seq
\seq_gpush:Nn \g_mho_example_seq {abc}
```

- `\seq_gpush:Nn` is a function from the seq module (sequences)
- `N` is single token, `n` is argument in braces.
- This globally pushes its second argument onto the global stack `\g_mho_example_seq`
- You can also pop:

  `\seq_gpop:NN \g_mho_example_seq \l_mho_target_tl`

Expansion control

As mentioned in other talks, expansion control is not trivial

- You have to know where to insert `\expandafter`
- You have to know your $2^n - 1$ table to insert the magic number of `\expandafter`

We use the argument signature to make this easier.

x means full expansion (with `\edef`), then pass on to `n`.

o means expand once, then pass on to `n`.

c means construct control sequence (with `\csname...\endcsname`), then pass on to `n`.

An example, cont.

- As before, but we first have to construct the name of the sequence.

  ```
  \seq_gpush:cn {g_mho_example_seq} {abc}
  ```

- Same result as before.
- No use of `\expandafter` or `\csname`.
- The code is much easier to read and maintain.
- `x` expansion:

  ```
  \seq_gpush:cx {g_mho_example_seq}
                { \tl_if_empty:nTF {#1} {empty}{#1} }
  ```

Renaming

- The first version of expl3 was fairly consistent in its naming
- But some parts needed a second go.
- `\def:Npn` → `\cs_set:Npn` and the set operation is now `\long`.
- `\let:NN` → `\cs_set_eq:NN`
- This way all data types have the operations `set`, `set_eq`, `new` and `new_eq`
- Token list pointers (tlp) changed name to just token lists (tl).
- Using tokens in the input stream is simple now: `\use_ii:nnn` is equal to `\@secondofthree` from LaTeX.

Retrieving value of a register

- Expansion control improved the situation a lot. Previously, you could do

```
\seq_gpush:No \g_mho_example_seq
              { \int_use:N \l_mho_magic_int}
```

- Worked, but required that you knew `\int_use:N` used exactly one expansion to return the result (because it is the `\the` primitive).
- But if adding from a different kind of container:

```
\seq_gpush:No \g_mho_example_seq
              { \l_mho_string_tl}
```

- So different syntax for different data types
- No error checking

Retrieving value of a register, cont.

Think about what you want rather than how!

- V for value of single token, v for the combination of c and V.
- The two examples from above then become

  ```
  \seq_gpush:NV \g_mho_example_seq \l_mho_magic_int
  \seq_gpush:NV \g_mho_example_seq \l_mho_string_tl
  ```

- No need to know how the data type is implemented
- — or how many expansions it takes to get to the value.
- This also provides error checking for malformed csnames, i.e., those with meaning `\relax`.

  ```
  ! Undefined control sequence.
  \exp_eval_error_msg:w ...erroneous variable used!

  l.15 \tl_set:Nv \l_tmpa_tl {g_oops_tl}
  ```

Defining functions

- In TeX a function has a parameter text (#1#2...)
- In LaTeX we have `\newcommand[`*num*`]{...}` but no delimited arguments
- With the functionality built into expl3 and the document level layer (xparse), you rarely need delimited arguments.
- The argument signature already tells how many arguments the function expects.
- So we use that information!

  ```
  \cs_new:Nn \mho_function:nnn {``#1,#2,#3''}
  ```

- You can still use the primitive parameter text. This is the same:

  ```
  \cs_new:Npn \mho_function:nnn  #1#2#3 {``#1,#2,#3''}
  ```

Conditional processing

New strategy

- Read arguments, and perform (complicated) tests
- Then return a state, e.g., true, false, error, ...

Then in the second step we take the state and then use it:

TF true state returns first argument, false state returns second:
`\foo_if_bar:nTF{⟨arg⟩}{⟨true⟩}{⟨false⟩}`

T true returns the argument, false returns nothing.
`\foo_if_bar:nT{⟨arg⟩}{⟨true⟩}`

F false returns the argument, true returns nothing.
`\foo_if_bar:nF{⟨arg⟩}{⟨false⟩}`

p returns boolean true or false. `\foo_if_bar_p:n{⟨arg⟩}`

Conditional processing, cont.

Here is a nice example from the boolexpr package, recently released to CTAN.

Simple task

- Take a single token argument, perform a test and then return one of two arguments following it, i.e., a `\foo_if_bar:NTF`⟨*arg*⟩`{`⟨*true*⟩`}{`⟨*false*⟩`}`
- If argument is one of `\the`, `\number`, `\dimexpr`, `\glueexpr` or `\muexpr` choose the true value, otherwise choose the false value.

Here is how it is done (look closely!)

Now what if I wanted a version that only returned the true value and returned nothing for false? `\foo_if_bar:NT`⟨*arg*⟩`{`⟨*true*⟩`}`

Conditional processing, cont.

Many ways to do this. The same but using the new expl3 interface.

```
\prg_new_conditional:Nnn \bex_test_Eval:N {TF,T}{
  \ifx#1\the                \prg_return_false:
  \else\ifx#1\number        \prg_return_false:
  \else\ifx#1\dimexpr       \prg_return_false:
  \else\ifx#1\glueexpr      \prg_return_false:
  \else\ifx#1\muexpr        \prg_return_false:
  \else                     \prg_return_true:
  \fi\fi\fi\fi\fi
}
```

This generates both `\bex_test_Eval:NTF` and `\bex_test_Eval:NT` but not the `F` and `p` variants.

Natural comparison

- Number comparison in TEX is tricky. Often you insert `\relax` many places to ensure scanning has stopped.
- They may stay behind in certain contexts!
- Natural is to ensure this happens automatically:
  `\intexpr_compare_p:nNn {5+3}<{2-\l_tmpa_int}`
- More natural is to remove most of the braces
  `\intexpr_compare_p:n {5+3 < 2-\l_tmpa_int}`
- Also supports <=, `!=`, >=.

Boolean expressions

- We now have a boolean expression parser
- Supports natural input syntax with
  - && for And
  - || for Or
  - ! for Not
  - () for grouping

```
\bool_if_p:n{
  \intexpr_compare_p:n {1=1} &&
  (
    \intexpr_compare_p:n {2=3} ||
    \intexpr_compare_p:n {4=4} ||
    \intexpr_compare_p:n {1=\error} % is skipped
  ) &&
  !(\intexpr_compare_p:n {2=4})
}
```

Summary

- All parts of expl3 have undergone revision
- No big changes expected – only extensions
- Appears in TeX Live 2009.
- Used in higher level modules (xparse, template) plus finding its way into other packages.