The LaTeX3 Programming Language
A syntax proposal for TeX macro programming

The LaTeX3 Project
latex-l@urz.uni-heidelberg.de

Abstract

This paper proposes a new set of programming conventions suitable for implementing large scale TeX programming projects such as LaTeX. (This syntax is not suitable for either document markup, or as a style specification language.)

The main features include a systematic naming scheme for all commands, and a systematic mechanism for controlling the expansion of arguments to functions.

The syntax is under consideration as a basis for programming within the LaTeX3 project.

This paper is based on a talk given by David Carlisle but describes the work of several people, principally: Frank Mittelbach, Denys Duchier, Rainer Schöpf, Chris Rowley, Michael Downes, Johannes Braams, David Carlisle and Alan Jeffrey.

1 Introduction

This paper describes a TeX based language which is intended to provide a more consistent and rational programming environment for the construction of large scale TeX macro projects such as LaTeX.

Variants of this language have been in use within the LaTeX3 project since around 1990 but the syntax specification to be outlined here should *not* be considered final. This is an experimental language, and the syntax and command names may (and probably will) change as more experience is gained with using the language in practice.

2 Programming Interface levels for LaTeX

One may identify several distinct languages that one might want to see in a TeX based system. This paper will *only* be concerned with the last of these three.

Document Markup This language consists of the commands that are to be embedded in the document instance. It is generally accepted that such a language should be essentially *declarative*. One might consider a traditional TeX based markup such as the LaTeX2 markup as described in [1], or alternatively one might consider an SGML based markup.

One problem with more traditional TeX coding conventions is that the command names and syntax of the TeX primitives are designed to have a 'natural' syntax when used directly by the author as document markup. In fact one almost never uses the primitives in this way, rather they are just used to define higher level commands.

Designer's Interface In order to easily translate a (human) designer's design specification into a program that accepts the document instance one would ideally like to have a declarative language that allows the relationships and spacing rules of the various document elements to be easily expressed. As this language is not embedded within the document text, it may take a rather different form to the markup language described above. For SGML based systems one may consider the DSSSL language as playing this role. For LaTeX2, then this level was essentially missing in LaTeX2.09. LaTeX 2_ε made some improvements in this area but it is still the case that implementing a design specification in LaTeX requires more 'low level' coding than would be desired.

Programmer's Interface This language, the subject of this paper, is the implementation language in which the basic typesetting functionality is implemented, building upon the primitives supplied by TeX (or a successor program). It may also be used to implement the previous two languages 'within' TeX, as in the current LaTeX system.

3 Programming language: Main Features

The language outlined in this paper aims to provide a suitable base for coding large (and small) scale

projects in TeX. Its main distinguishing features are the following.

- Consistent naming scheme for all functions, including TeX primitives.
- Standard mechanisms for controlling argument expansion.
- Provision of sufficiently rich set of core functions for handling programming constructs such as: sequences, sets, stacks, property lists, etc.
- White space ignored.

4 Naming Scheme

The name space is divided between *Functions* and *Parameters*. Functions (normally) take arguments and are executed, Parameters are usually passed as arguments to functions. They are not directly executed, but accessed though accessor functions.

Functions and parameters associated with a similar function (for example accessing counters, or manipulating lists, etc.) are arranged into *modules*. Before giving the details of the form of the command names, we give a few examples.

`\l_tmpa_box` is a local parameter (hence the `l_` prefix) corresponding to a box register.

`\g_tmpa_box` is a global parameter (hence the `g_` prefix) corresponding to a box register.

`\c_one` is the constant (`c_`) parameter with value one.

`\cnt_add:Nn` is the function which adds the value specified by its second argument to the count register specified by its first argument. The different natures of the two arguments are indicated by the `:Nn` suffix. The first argument must be a single token specifying the name of the count parameter. Such single token arguments are denoted `N`. The second argument is a normal TeX 'non-delimited argument' which may either be a single token, or a brace delimited token list containing an expression for the value to be added. The `n` denotes such 'normal' argument forms.

`\cnt_add:cn` is similar to the above, but in this case the the counter is specified in the first argument by a list of tokens that expands to the *name* of the count parameter.

These examples should give the basic flavour of the scheme. Parameters are classified into local, global or constant (there are further more technical cases in addition to these three) and this access type is shown by `\l_`, `\g_` or `\c_`. Functions are arranged by *module* (The `cnt` module in these cases) with a descriptive name followed by an indication of the type of argument to be passed.

In more detail the specification of the names is as follows.

Functions have the following general syntax:

\⟨*module*⟩_⟨*description*⟩:⟨*arg-spec*⟩

The programmer can chose an arbitrary ⟨*module*⟩ name (consisting of letters only) a group of functions with related functionality are then all given names prefixed by this module name. The ⟨*description*⟩ is a description of the functionality of the function, and should consist of letters, and possibly _ characters. ⟨*arg-spec*⟩ describes the type of arguments as will be described below.

The syntax of parameters is as follows:

\⟨*access*⟩_⟨*module*⟩_⟨*description*⟩_⟨*type*⟩

⟨*access*⟩ describes how the parameter can be accessed. The principal access types are *constant*, *local* or *global*. As described below, some special access types relate to TeX primitive parameters. The meanings of ⟨*module*⟩ and ⟨*description*⟩ in the parameter syntax is the same as that for functions. Finally ⟨*type*⟩ should denote the type of parameter, such as `cnt` for count registers, etc.

Typical ⟨*module*⟩ names in the kernel include `cnt` for integer count related functions, `seq` for functions relating to sequences, `box`, etc. Normally additional packages adding new functionality would add new modules as needed.

The ⟨*description*⟩ is an arbitrary name for the function or parameter, consisting of letters, or the _ character.

Function names always end with an ⟨*arg-spec*⟩ after a final colon. This gives an indication of the types of argument that a function takes, and provides a convenient method of naming similar functions that just differ by their argument forms, as will be explained below.

The ⟨*arg-spec*⟩ consists of a (possibly empty) list of characters each denoting one argument that the function takes. It is important to note that 'argument' here refers to the conceptual argument of the function. The top level TeX macro that has this name typically has no arguments. This is similar to the existing LaTeX convention where one says that `\section` has an optional argument and a mandatory argument, whereas the TeX macro `\section` actually takes no parameters at the TeX level, it merely calls some standard LaTeX internal functions which look ahead for star forms and optional arguments. The list of possible argument specifiers includes:

n Unexpanded token (or token-list if in braces). In other words this is a standard TeX undelimited macro argument.

o One time expanded token or token-list. In the case of a token list then only the first token in the list is expanded.

x Fully expanded token or token-list. Typically this means that the argument is expanded in the style of `\edef` (`\def:Npx`) before being passed to the function.

c A character string used (after expansion) as a command name. The argument (a token or braced token list) should expand to a sequence of characters which is then used to construct a command name (via `\csname`, `\cs:w`). This command token is passed as the argument to the function.

N A single token. (Unlike `n`, this argument must not be surrounded by braces). A typical example of a command taking an N argument is `\def`, in which the command being defined must be unbraced.

O Single unbraced token that is expanded once and passed (as a braced token list) to the function.

X Single unbraced token that is fully expanded and passed (as a braced token list) to the function.

C A character string used as for `c` arguments but the resulting command token is then expanded (as for `O`) and the result passed as a braced token list to the function.

p A primitive TeX parameter specification. This can be something simple like `#1#2#3` but may be arbitrary delimited argument syntax, such as `#1,#2\q_stop#3`.

T, F These are special cases of `n` arguments, used as the true and false cases in conditional tests.

D 'Do not use'. This special case is used for TeX primitives that are only used while bootstrapping the LaTeX kernel. If the TeX primitive needs to be used in other contexts it will be given an alternative name with a more appropriate argument specification.

w 'weird' syntax. Used for arguments that take non standard forms, usually delimited arguments that are needed internally to implement certain modules, and also the boolean tests of many of the primitive `\if...` tests.

For parameters, the ⟨*type*⟩ should be from the list of available data types (which include the primitive TeX registers, but also data types built within the system).

Possible values for ⟨*type*⟩ include:

cnt Integer valued counter.

toks Token register.

box Box register.

fcnt 'Fake' count register. A data type supplied by the kernel to avoid problems with the limited number of available count registers in (standard) TeX.

The ⟨*access*⟩ codes that are used in parameter names include

c Constants.

l Parameters that should only be set locally.

g Parameters that should only be set globally.

5 Checking Parameter assignments

One of the advantages of having a consistent scheme is that the system can provide more extensive error checking and debugging facilities. For example a function that makes a global assignment can check that it is not passed a local parameter as argument by checking that the name of the command to be assigned starts with `\g_`. Such checking is probably too slow for production runs, but the kernel has hooks built in to allow a format to be made in which all functions perform this kind of check. A typical section of code might look like

```
%<*check>
\def_new:Npn \toks_gset:Nn #1 {
   \chk_global:N #1
   \pref_global:D #1
}
%</check>
%<*!check>
\let_new:NN
  \toks_gset:Nn  \pref_global:D
%</!check>
```

The function `\toks_gset:Nn` takes a single token (`N`) specifying a token register, and globally sets it to the value passed in the second argument. So typical use would be

```
\toks_gset \g_xxx_toks {some value}
```

In the normal definition, `\toks_gset` can be defined just to be `\let` to `\global`, as the primitive token register does not require any explicit assignment function. This is the `%<*!check>` code above. However the alternative definition first checks that the argument passed as `#1` is a global parameter and raises an error if it is not. It does this by taking apart the command name passed as `#1` and checking that it starts `\g_`.

6 Consistent use of accessor functions

The primitive TeX syntax for register assignments has a very minimal syntax, and apart from box functions there are no explicit functions for assignment or use of the registers. This makes it very difficult

to implement alternative data types with a syntax that is at all similar to the syntax for the primitives, and also encourages a coding style that is very error prone.

As noted in the example given above, The LaTeX data types are provided with explicit functions for setting and using the parameters even when these have essentially empty definitions. This allows for better error checking as described above, and also allows the construction of alternative data types with a similar interface. For example the 'fake counter' data type mentioned previously works at the user level just like the data type based on primitive count registers, internally it does not use count registers though. Typical functions in the `fcnt` module include:

`\fcnt_new:N \l_tempa_fcnt`
Declare the local parameter `\l_tempa_fcnt` as a fake counter.

`\fcnt_add:Nn \l_tempa_fcnt \c_thirty_two`
Increment the counter by 32.

7 Expansion Control

Anyone who programs in TeX is used to the problem of arranging that arguments to functions are suitably expanded before the function is called. A couple of real examples copied from `latex.ltx`:

```
\global
  \expandafter\expandafter\expandafter
  \let
  \expandafter
       \reserved@a
           \csname\curr@fontshape\endcsname

\expandafter
  \in@
    \csname sym#3\expandafter\endcsname
        \expandafter{\group@list}%
```

The first piece of code is a global `\let`. The token to be defined is obtained by expanding `\reserved@a` one level. The command that it is to be let too is obtained by fully expanding `\curr@fontshape` and then using the tokens produced by that expansion to construct a command name. This results in the mess of interwoven `\expandafter` and `\csname` beloved of all TeX programmers, and code that is essentially unreadable.

A similar construction using the conventions outlined here would be

```
\glet:Oc
   \reserved_a: \l_current_font_shape_tlp
```

The command `\glet:Oc` is a global `\let` that expands its argument once, and generates a command name out of its second argument, before making the definition. This produces coe that is far more readable.

Similarly the second piece of code above produces a token list by expanding `\group@list` once, and then creates a command name out of '`sym#3`' (this is inside the definition of another fumction). The function `\in@` is called which tests if its first argument occurs in the token list of its second argument.

Again it would be much clearer, if the above function `\in@` was called (say) `\test_if_in:nn` (a function taking two normal '`n`' arguments) and then a variant function was defined with the appropriate argument types and simply called as follows:

```
\test_if_in:co {sym#3} \group_list:
```

Note that apart from the lack of `\expandafter` the space after `}` will be silently ignored.

For many common functions the kernel will provide functions with a range of argument forms, and similarly it is expected that extension packages providing new functions will make then available in the more common forms. However There will be occasions where it is necessary to construct such a variant form.

A consistent mechanism is provided by the kernel to produce functions with any argument type, starting from a function that takes 'normal' TeX delimited arguments. Suppose you have a function `\cmd:nnn` that takes two arguments, and you need to construct `\cmd:cnx` a variant form in which the first argument is passed as a *name* of a command, and the third argument must be fully expanded before being passed to `\cmd:nnn`.

One simply defines `\cmd:cx` as follows:

```
\def:Nn \cmd:cnx {\exp_args:Ncnx \cmd:nnn}
```

The function `\exp_args:Ncnx` takes as its first (`N`) argument the 'base' function, and then grabs the next three arguments from the token stream, acts on the first with `\csname`, and the last with `\edef` and then constructs a call to the base function with suitably transformed arguments. So

```
\cmd:cnx {abc}{pq}{\rst\xyz}
```

is equivalent, but emminently more readable, to

```
\edef\temp{\rst\xyz}
\expandafter\cmd:nnn
    \csname abc\expandafter\endcsname
     \expandafter{%
       \expandafter p\expandafter q%
     \expandafter}%
   \expandafter{\temp}
```

A large range of argument processing functions are provided in addition to `\exp_args:Ncnx`. If you need a particular argument combination for which a function is not provided, one may be constructed in a simple way. For example you need to construct `\exp_args:Nxcxcxc` a function that fully expands arguments 1, 3 and 5 of a given function, and produces commands to pass as arguments 2, 4 and 6 using `\csname`. The definition is simply

```
\def:Npn \exp_args:Nxcxcxc
         {\::x\::c\::x\::c\::x\::c\:::}
```

Similar functions, `::o` etc exist for all the other argument types, and they may be strung together in any order, terminated by `\:::` to create a function which processes arguments in the desired way.

As hopefully demonstrated, the use of variant forms greatly improves the readability of the code, and experience shows that the longer command names which result from the new syntax do not realy make the process of *writing* the code any harder.

8 The Current Experimental Distribution

The initial implementations of a TeX format using this kind of syntax were made with an unreleased (and non functional) format (which pre-dates LaTeX 2_ε!) The current distribution consists of a subset of the functionality of that format, converted to run as packages on top of LaTeX 2_ε.

The intention is to allow experienced TeX programmers to experiment on the system and to comment on the interface. This means that ***the interface will change***. No part of this system, including the names of any commands should be relied upon as being available in a later release. Please do *experiment* with these packages, but do not use them for documents that you expect to keep unchanged over a long period.

In view of the proposed experimental use for this distribution, we currently have only converted a few modules for use with LaTeX 2_ε. These set up the basic conventions, and then implement a few basic programming constructs such as lists and sequences. These are really to give a flavour of the code, and to indicate that the intention is that the kernel provide a sufficiently rich set of programming constructs so that packages may use them and thus more efficiently share code, unlike the situation in the current LaTeX where every large package implements its own version of lists, stacks etc.

The current packages are:

l3names Sets up the basic naming scheme, including naming the TeX primitives. If used with the option `[removeoldnames]` then the old primitive names such as `\box` are *undefined* and thus made available for user definitions. Use of this option might possibly break existing TeX code!

l3basics Some basic definitions that are used by the other packages.

l3chk Functions to check (and make) definitions (comparable to the existing `\newcommand` or `\renewcommand`)

l3tlp Token List Pointers. A basic LaTeX3 data type for storing token lists. (These are essentially macros with no arguments.)

l3expan The argument expansion module discussed in the previous section.

l3quarq A 'quark' is a command that is defined to expand to itself. So it may not be directly used (it would generate an infinite loop) but has many uses as special markers within LaTeX code.

l3seq A module implementing the basic list and stack data types.

l3prop Property lists are the data type for handling key/value assignments.

The distribution also contains the TeX source for this document, a docstrip install file and two small test files.

References

[1] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.