The `\noname` macros — a technical report

Jonathan Fine

Abstract

The `\noname` package provides a powerful environment for writing TEX macros. Its use makes macros easier to read, easier to write, and easier to document. It allows ready access to powerful control macros. It allows diagnostic and other code to be tagged for conditional inclusion. The `\noname` package is fully compatible with existing macros.

Here are two major features. It allows easy access to arbitary character tokens. Lines that do not begin with a white space character are comments, and are ignored.

The intention has been to provide the productive features that users of other programming languages take for granted. This article provides an outline of the history, design and implementation of the `\noname` package.

Acknowledgements

I would like to thank Nelson Beebe, and particularly Michael Downes, for their careful comments on an earlier version of this article.

1 Introduction

The `\noname` package grew out of work the author was doing two years ago. The goal was to write macros, for setting verbatim code, that would set source in a `\tt` font, and comments in a proportional font. This effect was to be achieved without additional mark up of the input file. Other refinements over the usual verbatim listing for source code were also desired.

In the course of this programming, extensive access to characters with `\catcode`s other than those usually given was desired. This proved to be a stumbling block for this project, which still awaits completion. Various programming tricks were introduced. The result of systematically developing these devices is the `\noname` macros. Although they have now reached the stage of being useful, there are developments being considered that will further increase their power and usefulness.

The physical activity of erecting a building commences with the digging of a hole, that will become the foundations that support the planned structure. The larger the building, the deeper the hole. The `\noname` package is intended to provide secure foundations for large scale collections of TEX macros.

2 Examples

Here is a line of code from `plain.tex`. It supports the `\newif` construction. It creates a control sequence `\if@` that must be followed by *other* characters '`i`' and '`f`' with catcode *other*, i.e. 12. Such funny letters arise from use of `\string`.

```
{\uccode`1=`i \uccode`2=`f
  \uppercase{\gdef\if@12{}}}
```

(The purpose of `\if@` is to extract the string `foo` from `\iffoo`, which is then used to construct `\foofalse` and `\footrue`. The macro `\if@` is also intended to give an error if the argument to `\newif` is *not* of the form `\if...`).

Here is the same macro defined using `\noname`.

```
\def \if@ 'i'f {}
```

The right quote symbol '`'`' is an *escape character* that serves to produce a character with catcode *other*, whose character code is given by the following alphabetic constant.

Here is another example. The `\noname` macro definition

```
\def\spaces{ ~~~~~ }
```

defines `\spaces` to be a macro whose replacement text is five ordinary space tokens. (Ordinarily, special tricks are required to get a space after a control word or another space). Finally,

```
\def !\^^M { \par }
```

will in `\noname` define active carriage return to expand to `\par`.

3 Influences

This section attempts to list the various sources for the design of the `\noname` macros.

3.1 Knuth's `WEB` system. `WEB` allows source and documentation to be mixed in the same file, and in a disciplined manner. It implements literate programming. The `WEB` system allows a module to be incrementally coded. This means, to use an example of Knuth, that with a module named ⟨*Global variables in the outer block*⟩ one can add to this module as, where, and when new global variables are required. Without this feature the global variables would have to be declared all together and at once in the PASCAL source file. Such a feature is not provided, nor planned, for `\noname`.

Perhaps the most important idea borrowed from `WEB` is the introduction of a preprocessor to augment the facilities of the language.

Because Pascal is not well adapted to storing character strings, Knuth implemented a "string pool" feature. Such a device might be welcome when writing TeX macros, for storing error and other messages, which otherwise would consume large amounts of main memory. These messages could be stored either in in TeX's string pool, or in separate files on disc, to be read in as needed.

3.2 The *C* programming language. This language is used widely for both system and application programming, and its syntactic style is widely known and imitated. Source written using `\noname` tends to have a *C*-like appearance.

The *C* language provides a preprocessor for source files, just as does `WEB`. The `\noname` package also includes a source file preprocessor. In particular, the hash command syntax has been copied from *C*, although the functioning will be different. Also copied is the use of the colon '`:`' as a label, and names for some of the control macros. (The MS-DOS batch language also uses '`:`' as a label).

3.3 Mittelbach's `doc` option for LaTeX. This work showed to me that documenting source files for TeX macros was a problem. Although his solution is a significant advance, it has limitations. His open recognition

> [t]he method of documentation of TeX macros which I have introduced here should ... only be taken as a first sketch.
>
> *TUGboat* 10, no. 2 (1989), page 246

of this encouraged me to find my own solution.

The convention — that the initial character determines whether a source file line is a comment or code or a hash command — came from a wish to have a natural input scheme for the typesetting of source files.

Although the `doc` option gives a pleasant appearance to the large comment blocks, it leaves the macro language of TeX unchanged, and sets small comments within the code lines in a typewriter font.

`WEB` consists of `TANGLE`, which provides a language enhancement to PASCAL, and `WEAVE`, which typesets source files. The `doc` system as published provided no language extension. It leave the TeX macro language unchanged. (The later `docstrip` feature does, however, allow for code to be tagged to conditional inclusion).

`WEAVE` recognises the key words and symbols of PASCAL, and uses this when the source file is typeset, to improve the typographic quality. When `doc` typesets a source file it sets code lines verbatim. Apart from recognizing and indexing control words, it has no understanding of the TeX macro language. (The `\noname` package, by counting braces, is able to guess when a `\def`-inition has come to an end).

As mentioned earlier, much of the motive for `\noname` came from a wish to code something superior to the `doc` option. This turns out to be a larger project than I imagined. The `\noname` macros provide part of the foundation.

3.4 Smalltalk. Certain concepts in Smalltalk have had an influence on the internal coding of `\noname`. The idea of compiling source into an intermediate form came from Smalltalk, which uses *bytecode instructions*. It also provides an excellent example of a productive integrated programming environment.

3.5 Wirth's Modula-2. When the module concept is added to `\noname`, it is quite possible that the syntax, and some details of the implementation, will draw upon Modula-2.

4 Design and Implementation

4.1 The Basic Problem. A TeX macro consists of a string of tokens. A token is either a character token — i.e. a character/catcode pair — or a control sequence. The problem of producing any given macro therefore reduces to producing any given control sequence, and any given character token. The control sequence problem shall be put to one side, for except when control sequence names are used to store textual information, it is enough that the control sequence have a comprehensible name formed from a fixed collection of characters.

Arbitary character tokens are produced via careful use of the `\uppercase` command. They are placed into macros by use of the `\aftergroup` primitive. *The TEXbook*'s "dirty trick" construction of a macro whose replacement text is `\n` asterisks (p373–4) illustrates the basic technique.

4.2 `\load`, `\comp`, `\online` and `\hsl`. The macro writer will specify, using a syntax to be described later, a sequence of control sequences and character tokens, to be formed into a macro. The `\noname` macros will read and act upon this input stream — there are examples above — so as to construct the desired macro.

The `\load` command will read these macro-building instructions from a file, and place the result into TEX's memory. The `\comp` command will write the content of these instructions to a file on disc, in a form that can then be re-processed at high speed, by use of the `\hsl` command. Finally, the `\online` command is like `\load`, except that it takes its input from the console, rather than a file.

5 Structure of source files

5.1 White space. All white space is ignored (unless preceded by a `\noname` escape character). It is no longer necessary to use '`%`' symbols to prevent space tokens creeping into your macros.

5.2 Comment lines. Any line that does not begin with white space is a comment line, and will be ignored (unless it begins with a '`#`' hash character — see below).

5.3 Escape characters. The `\noname` package has a rich range of escape characters.

- `~` produces an ordinary space token.
- `'` produces a character, with catcode *other* = 12, whose character code is given by the token immediately following '`'`'. This token may be a white space token, or some other character, or a control symbol.
- `!` is like '`'`', except that it produces an *active* character.
- `|` is the *bar* construction, which allows access to arbitary character tokens. It should be followed by the `\catcode`, as a hexadecimal digit, and then the character code, as a character or a control symbol. Thus `|D` is equivalent to `!` and `|C` is equivalent to `'`.
- `:` is a label which produces an otherwise inaccessible macro, whose expansion is empty. This device is most useful when used in conjuction with the *Basic Control Macros* cited elsewhere.
- `@* ;` will produce unusual character tokens. They are intended for use with the `\CASE` and `\FIND` macros (*TUGboat*, to appear), and some other purposes.

Finally, the characters `{}$#^_&` and `%` have their usual effect.

5.4 Control words. When using `\noname`, not only can the letters `a..zA..Z` and the `@` character be used for forming control words, but also the characters `$&*_:` and the digits `0..9`. This does not interfere with the usual use of the characters, outside of control words. For example

```
\def\subscript_character{ _ }
```

defines `\subscript_character` to be a macro whose replacement text is a *subscript* character, with '`_`' as its character code.

5.5 Numeric constants. Within `plain` and LATEX the control sequence `\m@ne` is used to refer to a `\count` register whose value is fixed to be −1. This feature is provided because −1 is a ubiquitous constant. Macros run quicker, and occupy less space, if `\m@ne` is used in place of `-1`. The `\noname` package provides the same functionality, but by typing `[-1]`.

With `\noname`, the tokens `[nnn]` where `nnn` is a literal number such as `-1` or `"57` or `16` will produce a control sequence which is to store the number `nnn`. This allows the popular numeric constants to be referred to in a literal manner, rather than via cryptic names.

A similar convention applies to numeric constants specified as character constants. The character '`` ` ``' followed by a *control symbol* such as `\x` or `\^^M` will produce a control sequence which stores a number, namely the ASCII value of `x` or `^^M` respectively.

If `[` is followed by a token that cannot begin a literal number, i.e. other than `0..9` or `+-'"`, then no special behaviour occurs. Similarly, if `` ` `` is not followed by a control sequence, then no special behaviour occurs.

5.6 Hash commands. This is a feature borrowed from *C*. Any source line beginning with a hash `#` is a hash command. Hash commands control the processing of the file. They allow conditional inclusion of code. For example, if `\ifdebug` is `\iftrue` then the line below

```
#\ifdebug
                \checkingcode
#\fi
```

which contains `\checkingcode` will be processed, while if `\ifdebug` is `\iffalse` then this line will be skipped.

This feature allows the same file to contain several variants of the same code. Currently, LaTeX has three files — `art10.sty`, `art11.sty`, and `art12.sty` — which are identical in all aspects, except for the values of some numerical and other parameters. Use of hash commands allows these files to be described using a single source file.

This feature can also be used to maintain a single file for several versions of a macro package.

6 Control macros

The author's *Basic Control Macros* in *TUGboat* 12, no. 2 are easier to use within the `\noname` environment, for they depend on a label '`:`' being available. The author has also written powerful control macros `\CASE` and `\FIND`, which again depend on `\noname` features — in this case that `*` and `;` produce not ordinarily accessible character tokens.

The author is about to release a control macro `\FSA` (for Finite State Automaton). Here is an example of its use. When, on a page, one vertical item is placed beneath another, vertical space may be required, or perhaps a penalty, or some other activity. The `\FSA` device allows the decision table for such transitions to be coded in a simple, elegant, and economical manner. It will use `@*;` as delimiters.

7 Structure of .hsl files

The details of the fine structure of the `.hsl` files should be of little concern to macro programmers, so long as it adequate to support their needs.

Here is an extract from a `.hsl` file.

```
^^@ \FILE tutor.hsl %
^^@ \year 1992 \month 9 ...   720 %
%{ \hsl"{"c"o"n"t"r"  ...  %%%%%%%
% \tracinglostchars  ...  "{"m"a"i%
%"n"m"e"n"u"}"}}{   ...  e \online%
% \def \online"{ \e  ...  #'1"{%%%%
% \immediate \write"\  ...  "{%%%%%
```

Normally, during a `\hsl`, `^^@` is a comment character and `%` is ignored. This is designed to support macro library files. (In this context, see *The TeXbook*, p382-4). By setting `%` to comment, the macros can be skipped at high speed.

By setting `^^@` to ignore, it is possible to read the `\FILE` and date information in the header. The date can be used for version control and compatibility. For example, by storing multiple versions of the same macros in a single file, the most recent first, it is possible to load the macros that are in force at some given date. Simply load the first macro package whose date is before the given date. (The date is precise to time of day, in minutes, as supplied by TeX's `\time` primitive).

Currently, the optional code controlled by hash commands can be used only to generate multiple `.hsl` files, each obtained by processing a different subset of the source file. Essentially, the variant is determined at the time of the `\comp`-ile. However, the basic structure of the `.hsl` file is sufficiently rich, as to allow these variants to be combined into a *single* `.hsl` file. The setting of a flag will then control the choice of variant *at the time of the* `\hsl` *of the file.* This feature could be used to produce, if wished, a single `art.sty` file for use with LaTeX.

8 Modularity and named parameters

Many other languages restrict the scope of an identifier, so that the same identifier can be used for different purposes in different contexts. For example, in *C*, identifiers declared within a function are local to that function, while identifiers prefixed by the keyword `static` are local to the file in which they appear.

TeX has a single global name space. Consequently, each author of macros has to be sure that his or her control sequences do not clash with those of `plain`, LaTeX, or some other package. This is a burden.

Here is a related matter. In other languages the parameters to functions (the TeX equivalent is macros) are identifiers. This improves the code greatly. For example

```
\def\centerline #\text
{
  \line{ \hss \text \hss }
}
```

is easier to read and maintain.

The addition of these facilities to the `\noname` package is being investigated.

9 Performance

The single most important aspect of the performance of the `\noname` package is the degree to which it allows the macro writer to produce better code quicker. I will leave the measurement of this to others, who are able to be more objective. My experience of using `\noname` is that the code written is much easier to read after the event, and that the various helpful facilities reduce coding time by between 10 and 30 percent. The saving will depend on the nature of the macros being written, and the extent to which they are basic. If two (or more) almost identical versions of the same file are required, then the time saving can be much greater. This is also true if active and other special characters are required.

The commands `\load` or `\online` are quick enough, on a slow machine, to process small (say 50 line) files, but become tedious for much larger files. They also have an overhead of one (1) control sequence for every new control sequence processed. This overhead will increase, and the performance fall, when the various enhancements are added. However, the `\hsl` command works at much greater speed, and with minimal overhead.

The `.hsl` files are much smaller that the source files from which they are generated. They can be combined into a single library `.hsl` file, with conditional run-time loading of the constituent parts. These features can be used to save mass storage requirement (note that file space is allocated in fixed size blocks), and reduce traffic on a network and on `email`. Note also that it can take the operating system longer to find a small macro file, than it takes TeX to process it.

However, most macros are loaded once, and then `\dump`-ed as a format file, which can then be loaded at high speed. The quality of the code will then determine the size of this file, and thus how quickly it can be loaded.

10 Future developments

Briefly, here are some projects that are under way. Much progress has been made on coding a *pretty printer* for typesetting source files written in the `\noname` dialect. It is intended that it should also be able to typeset suitably laid out *C* and *C*++ source files.

A proof-of-concept prototype for a *single step debugger*, that will execute or expand TeX macros and commands one at a time, has been coded, and will also form part of the `\noname` package. I hope that it will be useful both for learning and teaching how TeX works, and also for development.

Finally, an interactive tutorial for `\noname` — consisting of TeX macros and so running within TeX — has been written.

11 Availability

This package has been developed privately. Future developments will require financial support, most likely from sale of the software.

Publishers and other major users of TeX require custom macro packages. These are either written in house by expert staff, or commissioned from outside. These packages are usually proprietary, although publishers tend to make them available when appropriate to their authors.

At the other end, there is a large mass of unsupported macro files, of variable quality, available for no cost. In addition, there are packages such as `plain`, LaTeX, PiCTeX.

Discussion with TeX users will reveal the technical and other merits of `\noname`, and help provide a basis for pricing, licensing, distribution and other policies.

The current version is already useful. It may take six months to add modules, named parameters and other advanced features. Also required, as in *C*, are libraries of standard functions.

A demonstration version is available (from the author only), so long as you agree to respect his intellectual property rights.

◇ Jonathan Fine
203 Coldhams Lane
Cambridge
CB1 3HY
England
`J.Fine@pmms.cam.ac.uk`