Case changing: From TEX primitives to the Unicode algorithm

Joseph Wright

1 Introduction

The concept of letter case is well established for several alphabet-based scripts, most notably Latin, Greek and Cyrillic. Upper- and lowercase[1] are so widely used that it may not be obvious that there are several subtleties in converting case. However, those subtleties are important in supporting a wide range of users, and getting all of them right is non-trivial.

Whilst the English alphabet has simple case-changing rules, when we look beyond English and (possibly) beyond the Latin alphabet, tracking the requirements becomes more complicated. Many of these have been codified by the Unicode Consortium, and following these guidelines means that different pieces of software can give consistent outcomes.

Here, I want to look at how case changing can be set up in TEX, primarily focussing on tools that the LATEX Project have provided, but in the wider context of the TEX ecosystem.

2 Different kinds of case operation

To understand what functionality is needed for case changing, we first have to know what types of input we might be dealing with. Broadly, there are two:

- Text: material that we will want to typeset or similar, and which contains natural language content. This material might also have some formatting, and may be marked up as being in a particular language.
- Strings: material used in code, for example as identifiers, to construct control sequences or to find files. This material will never have formatting, and should always give the same outcome, irrespective of the language in which a document is written.

Unsurprisingly, case changing strings is a lot more straight-forward than case changing text: there is no context to worry about. However, neither text nor string case changing is reversible, and that leads us to the different types of case changing operations that are needed.

The Unicode Consortium describe four case operations, three case mappings and one case folding:

- Uppercasing
- Lowercasing
- Titlecasing: changing the first character to uppercase and the remaining characters to lowercase — we will see later how a small number of situations need special handling
- Case folding: removing case information from the input to allow 'caseless' operations

(I strongly recommend the Unicode Consortium's FAQ at `https://unicode.org/faq/casemap_charprop.html` for more on the concepts here.)

Upper-, lower- and titlecasing material is about *text*: material to be read by people and which can have context- and language-dependence. In contrast, case folding is about computers: doing things mechanically for comparing internal information. Commonly, programmers use all-lowercase for that, but there are places where that would be wrong: again, we'll see some examples below. We'll also see that for TEX use, there are places we need 'programmer's upper- and lowercase (a.k.a. CamelCase) for strings.

3 Changing tokens or changing output

Before we look at the methods we can use in TEX to change case, it's worth bearing in mind that for *typesetting*, there's the possibility of leaving any change in appearance to the font mechanisms. Whilst this is complicated in classical TEX, LuaTEX offers the potential to delegate the task to well after character tokens have been handed to the paragraph-builder.

However, it's quite natural in many programming languages to change the case of text (referred to in many languages as *strings*). As such, I will focus on methods that can alter the characters in the TEX input stream.

4 Built-in TEX mechanisms

In the TEX world, the primitives `\lowercase` and `\uppercase` are the obvious starting point for case changing. These primitives use data stored in TEX using the `\lccode` and `\uccode` primitives, respectively, and only ever perform context-independent conversion. Whilst there are important uses for this behaviour well beyond 'normal' case changing, I am going to focus here only on their utility (or otherwise) for application to text.

The most obvious limitation of the primitives is that they assume a single mapping for all characters. Whilst there are a large number of simple relationships, there are exceptions: see the Unicode data file `SpecialCasing.txt` for the full list! The simple approach taken by the primitives means that there is no chance of handling context-dependence: this is most obvious with Greek, where there are two forms of lowercase sigma, one used only at the end of words (ς).

[1] The spelling of these concepts is somewhat variable: 'upper case', 'upper-case' and 'uppercase' are all valid. The LATEX Project have chosen the latter form as it makes creating clearly-named code functions easier!

At the TEX level, the other issue with the primitives is that they work by execution not expansion. I think almost every trainee TEX programmer will at some stage have tried

```
\edef\foo{\lowercase{STUFF}}
```

and been very surprised that it fails, and that they need to use

```
\lowercase{\edef\foo{STUFF}}
```

instead. This shows up when you want to change case inside a `\csname` construct too: you can't use `\lowercase` within the construction, but rather have to use it around the entire thing.

```
\lowercase{\csname FooBar\endcsname}
```

There's also one other very important consideration: the primitives convert character tokens only, but do not know the meaning of these tokens. This leads to a few issues:

- They do not convert text hidden in macros: relatively easily remedied by applying `\edef` (or `\protected@edef`) prior to using the primitive.
- They cannot handle letter-like control sequences such as `\aa` or `\l`.
- They cannot handle multi-byte 'letters' in 8-bit engines (so for example `\uppercase{é}` fails).
- They provide no mechanism for excluding characters from case changing, most critically those inside math mode.

5 The LaTeX 2ε kernel mechanism

The LaTeX 2ε kernel builds on the primitives to address some of the issues above. First off, these commands include an internal `\protected@edef`, which ensures that input is expanded first, then the case change is applied. They also provide definitions for a set of letter-like commands to allow them to be case-changed, stored as `\@uclclist`. This for example allows `\aa` to be uppercased to `\AA`.

6 The textcase mechanism

Case changing is fundamentally something that applies to *text*, and never to *mathematics*. In LaTeX, it is pretty clear which content is mathematical, and David Carlisle's textcase package makes it possible to case change text containing math mode content without 'breaking' the latter. Thus for example

```
\MakeTextUppercase
  {A simple formula:
    $y = mx + c$}
```

yields

A SIMPLE FORMULA: $y = mx + c$

The package also allows text to be marked as not to be altered during case changing, using the marker command `\NoCaseChange`.

It's possible to load textcase such that it replaces the LaTeX 2ε kernel case-changing commands with its own. That gives away that it works in a fundamentally similar way: it's a more sophisticated wrapper around the TEX primitives. That means it still has the core issues of not knowing the meaning or context of its argument tokens, and not being usable in an expansion context.

7 The expl3 mechanisms

7.1 The core concepts

At the core of the expl3 case changing mechanisms is the idea that the implementation should provide, as far as possible, the full set of Unicode Consortium functionality. The code is also written to work purely by expansion, meaning that it *can* be used inside `\csname` construction or inside an `\edef`.

To support all this, the input must be examined on a token-by-token basis and converted 'manually'. It also means that `\lowercase` and `\uppercase` cannot be used. Instead, for single-token conversion, expandable functions which can convert single tokens are defined: `\char_lowercase:N`, `\char_uppercase:N`, `\char_titlecase:N` and `\char_foldcase:N`. Almost always, those are too low-level. Thus we will not look further at the 'back end', but will rather concentrate on functions which can be used for longer pieces of input.

7.2 Strings

In TEX terms, a string is a series of characters which are all treated as 'other' tokens (except spaces, which are still spaces). That's important here because it means strings won't contain any control sequences, and because with pdfTEX there can't be any (useful) accented characters.

The most obvious need to handle case in programming strings is when comparing in a caseless manner: 'removing' the case. Programmers often do that by lowercasing text, but there are places where that's not right. For example, as mentioned above, Greek has two forms of the lowercase sigma (σ and ς), and these should be treated as the same for a caseless test. Unicode defines the correct operation: case *folding*. In expl3, that's called `\str_foldcase:n`:

```
\str_foldcase:n { AbC }
```

gives

abc

whilst the slightly more challenging

```
\str_foldcase:n { ΌΔΥΣΣΕΎΣ }
```

gives

ὀδυσσεύσ

Much more rare is the need to upper- or lowercase a string. Unicode does not mention this at all, but in TEX we might want to construct a control sequence dynamically. To do that, we might want to uppercase the first character of some user input *string*, and lowercase the rest. We can do that by combining `\str_uppercase:n` and `\str_lowercase:n` with the `\str_head:n` and `\str_tail:n` functions:

```
\str_uppercase:f { \str_head:n { someThing } }
\str_lowercase:f { \str_tail:n { someThing } }
```

which produces

Something

(We could also split off the first token and use the single-character `\char_uppercase:N` here.)

7.3 Text: basics

Case changing text is much more complicated because it has to deal with control sequences, accents, math mode and context. The first step of case changing here is to expand the input as far as possible: that's done using a function called `\text_expand:n` which works very similarly to the LaTeX 2_ε command `\protected@edef`, but is *expandable*. We don't really need to worry too much about this: it's built into the case changing system anyway.

Upper- and lowercasing is straight-forward: the functions have the natural names `\text_uppercase:n` and `\text_lowercase:n`. These deal correctly with things like the Greek final-sigma rule and (with LuaTEX and XeTEX) cover the full Unicode range. Thus we can have examples such as the following. (Recall that spaces are ignored in expl3 input, and ~ is used to produce a space.)

```
\text_lowercase:n { Some~simple~English }
\newline
\text_uppercase:n { Ragıp~Hulûsi~Özdem }
\newline
\text_lowercase:n { ΌΔΥΣΣΕΥΣ }
```

some simple english
RAGIP HULÛSI ÖZDEM
ὀδυσσεύς

A variety of standard LaTeX accents and letter-like commands are set up for correct case changing with no user intervention required.

```
\text_uppercase:n { \aa{}ngstr\"{o}m ~ caf\'{e} }
```

produces the token list

```
\AA{}NGSTR\"{O}M CAF\'{E}
```

7.4 Case changing exceptions

There are places that case changing should not apply, most obviously to math mode material. There are a set of exceptions built-in to the case changer, and that list can be extended: it's easy to add the equivalent of `\NoCaseChange` from the textcase package. First, create and activate the command, excluding it from expansion and excluding its argument from case-changing:

```
\cs_new_protected:Npn \NoCaseChange #1 {#1}
\tl_put_right:Nn
  \l_text_case_exclude_arg_tl
  { \NoCaseChange }
```

then we can use it

```
\text_uppercase:n
  { Hello ~ $y = max + c$ }
\newline
\text_lowercase:n
  { \NoCaseChange { iPhone } ~ iPhone }
```

which gives us the desired results

HELLO $y = max + c$
iPhone iphone

without having to worry further. Note that case changing does take place within braces (in contrast to BibTEX's approach):

```
\text_uppercase:n { { Text } ~ More }
```

gives

TEXT MORE

The reason is simple: braces are difficult to control and to remove, and can lead to undesirable impact on kerning and so on. (It is likely that a dedicated conversion function to approximate BibTEX case protection by expl3 protection will be added; the biblatex maintainers are keen to have this ability.)

7.5 Titlecasing

Commonly, people think about uppercasing the first character of some text then lowercasing the rest, for example to use it at the start of a sentence. Unicode describes this operation as titlecasing, as there are some situations where the 'first character' is handled in a non-standard way. Perhaps the best example is `IJ` in Dutch: it's treated as a single 'letter', so *both* letters have to be uppercase at the start of a sentence. There are also a small set of codepoints that *look* like two letters, and have special forms when they appear as the titlecase first-character of a word.

Depending on the exact nature of the input, we might want to uppercase the first 'character' and then lowercase everything else, or we might want to uppercase the first 'character' and then leave everything

else unchanged. These are called `\text_titlecase:n` and `\text_titlecase_first:n`, respectively. Most of the time, things look pretty simple:

```
\text_titlecase:n { some~text } \newline
\text_titlecase:n { SOME~TEXT } \newline
\text_titlecase_first:n { some~text } \newline
\text_titlecase_first:n { SOME~TEXT }
```

gives

Some text
Some text
Some text
SOME TEXT

To see titlecasing in action, let's stick with Dutch (we'll see how the language argument works shortly):

```
\text_titlecase:nn { nl } { IJSSELMEER }
```

should be

IJsselmeer

and not what most English speakers might expect

Ijsselmeer

As we are not simply grabbing the first token of the input, non-letters are ignored and the first real text character is case-changed. So when we say

```
\text_titlecase:n { `some~text' }
```

we of course want the result to be

'Some text'

without needing to worry about that first character.

7.6 Language-dependent functions

One important context for case changing text is the language the text is written in: there are special considerations for Dutch, Lithuanian, Turkic languages and Greek. That's all handled by using versions of the case-changing functions that take a second argument: a BCP 47 string which can determine the path taken. We've already seen Dutch, so let's examine the other special situations.

We've seen Greek for handling the final-sigma rule, but we also need to remove accents when uppercasing (but *not* when titlecasing). Sticking with the hero of myth

```
\text_uppercase:n { Ὀδυσσεύς } \newline
\text_titlecase:n { Ὀδυσσεύς } \newline
\text_uppercase:nn { el } { Ὀδυσσεύς } \newline
\text_titlecase:nn { el } { Ὀδυσσεύς }
```

gives us the expected

ὈΔΥΣΣΕΎΣ
Ὀδυσσεύς
ΟΔΥΣΣΕΥΣ
Ὀδυσσεύς

For Turkish, it's all about the dot over an `i`: that's a different letter from the dotless ı, and so

```
\text_uppercase:n { Ragıp~Hulûsi~Özdem }
\newline
\text_uppercase:nn { tr } { Ragıp~Hulûsi~Özdem }
```

produces

RAGIP HULÛSI ÖZDEM
RAGIP HULÛSİ ÖZDEM

Finally, Lithuanian needs us to retain dots on `i` and `j` when we add certain accents

```
\text_lowercase:n
  { ÌÍĨI`I´I~J`J´J~Į`Į´Į~ }
\newline
\text_lowercase:nn { lt }
  { ÌÍĨI`I´I~J`J´J~Į`Į´Į~ }
```

produces (in `\large` for better visibility)

i̇̀i̇́i̇̃ı̀ı́ı̃ j̇̀j̇́j̇̃ į̇̀į̇́į̇̃
i̇̀i̇́i̇̃ ı̀ı́ı̃ j̇̀j̇́j̇̃ į̇̀į̇́į̇̃

with explicit 'more above' dot accents (and yes, it does look odd here, but that's a font shaper decision).

There's one more special case: the capital *Eszett* in German. Normally, capitalisation of ß (U+00DF) yields simply `SS` (two characters), but there is now a Unicode codepoint (U+1E9E) for the capital form, ẞ (that's a capital Eszett in the Courier Narrow font we're using in this article; it looks nearly the same as the lowercase). There is currently no official BCP 47 name for this (that I know of), so one needs to use

```
\text_uppercase:nn { de-alt } { ß }
```

8 The Lua route

There is another way to do case changing in expansion contexts: using Lua.

```
\directlua{tex.print(unicode.utf8.lower
  ("A string"))}
```

That works, but it leaves us with a few issues. For a start, anything that depends on TEX tokens, like math mode, needs more work. Secondly, it doesn't deal with things like Greek final-sigma. That's all solvable with enough Lua, but it's not (yet) an out-of-the-box solution.

9 Conclusions

Implementing all of the subtleties of case-changing following the Unicode Consortium requires effort, but is accessible and can be done by pure expansion. Case-changing functions in expl3 are now mature and stable, and ready for wider use.

◇ Joseph Wright
Northampton, United Kingdom
joseph dot wright (at) morningstar2.co.uk