Joseph's Adventures in Unicodeland

Joseph Wright

1 Unicodeland

The rich variety of human language has over time led us to a plethora of ways of writing down our communications. Historically, computer systems handled this poorly. The 26 letters of the English alphabet have defined the landscape of computerized text storage, first through the ASCII standard and later as the basis of many 8-bit systems. Over roughly the past quarter-century, this situation has begun to change with the development of the Unicode standard (The Unicode Consortium, 2015). Unicode takes us beyond the limitations of the 7- or 8-bit world into a much richer environment. In this 'Unicodeland' every character has its proper place, and every character can and should be correctly handled by compliant software.

The Unicode Consortium have defined not only a rich (and expanding) set of characters (or more accurately *code points*) to handle this variety of data, but have also explored and defined how these should be manipulated under a range of transformations. As befits a standards organisation, the Unicode Consortium have not tied this information to any particular implementation. Rather, they provide a set of machine-readable files and documentation guidelines to allow compliant implementations to be built for the range of tasks we use computers for.

In the TEX world, we have today two Unicode-capable engines in general use: XETEX and LuaTEX. Following a careful TEX tradition, these engines do not hard-code Unicode data or behaviours into the engines. Rather, they allow us to handle Unicode input and to control behaviours by reading in the appropriate data.

Here, I will look at two areas where we need to get Unicode data into the engine: setting up TEX's codes correctly for the Unicode range, and implementing case-changing. Whilst the focus here is on how we are tackling these problems for LATEX, the ideas should apply to all TEX users.

2 Setting up characters

As XETEX and LuaTEX can accept input across the full Unicode range they need to know how to treat a much greater number of characters than classical TEX engines. For example, with an 8-bit engine we usually restrict 'letters' for creating control word names to `A`–`Za`–`z`. With a Unicode engine that's not reasonable: anything that is a letter according to the agreed standard can and should be set to category code 11. However, what we don't want to do is code all of that in by hand. Luckily, all of the core Unicode data files are provided in plain text format (and indeed are written in ASCII) and are machine-readable. 'All' we have to do is parse the appropriate files as part of the TEX run.

The details of course are a little more complex. It turns out that we want to set up several things

- `\catcode`
- `\lccode`
- `\uccode`
- `\Umathcode`
- `\XeTexintercharclass` (XETEX only)

for all appropriate characters. To do that, we need to first work out which Unicode data files have the relevant information in them and then to parse it into a usable form.

As the Unicode Consortium deal with data for many purposes, it is not surprising that things like TEX's `\catcode` concept don't feature directly in the data files. Instead, we need to make some systematic decisions about relating Unicode properties to TEX. Most of this work was done by Jonathan Kew when he first released XETEX; at that time, he created a Perl script (`unicode-char-prep.pl`) to do the work (Kew, 2015). The LATEX team has now created a very similar script, using pdfTEX rather than Perl for the parsing but retaining most of the logic.

Much of the conversion is relatively obvious. Thus for example characters described by the Unicode Consortium as falling into one of the letter types are mapped to `\catcode` 11. However, there are other characters that need to be `\catcode` 11: combining marks and East Asian ideographs. Picking these up requires a bit of thought: the details of parsing the source files are more technical than conceptual. Reading all of the data with pdfTEX takes a few seconds, but luckily this only has to happen on the machine of one of the members of the team. For users, the processed file can be read *very* quickly: indeed, as part of format-building, it's negligible.

What the team has added in this area is setting up a single file to be read by both XETEX and LuaTEX, with the necessary conditionals inside the file. That means that the common outcomes are the same in all cases, with just the *additive* part for XETEX covering `\XeTexintercharclass`. Having the file provided by the team also means that it will be updated as part of wider kernel changes, which should provide some regularity as to when this takes place.

3 Case changing

3.1 The background

TEX provides us with two primitives for case changing, `\lowercase` and `\uppercase`. The logic behind these is simple: they convert single characters from one case to another based on the idea of one-to-one relationships. That's fine when we have a simple situation with two cases, one language and everything mapping neatly. This is, of course, TEX's background: for English, `\lowercase` and `\uppercase` are entirely reasonable.

Life gets more complicated once we introduce more variation. First, even apparently simple one-to-one relationships can be language-dependent. Perhaps the most obvious example is Turkish, where the upper case equivalent of `i` is `İ`, not `I`. Second, there's no context-dependence available: mappings are not always the same for the same characters. The Greek 'final sigma' rule is perhaps the best known of these situations: the correct lower casing of `ΌΔΥΣΣΕΎΣ` for example is `όδυσσεύς`, using the two different lower case sigma characters in Greek. There is then the question of the one-to-one mappings themselves: `fußball` in upper case is `FUSSBALL` with an extra character. It's clear from these issues (and other subtleties) that a more nuanced approach is needed.

In practice, making everything work with Unicode input requires a Unicode engine, so the ideas here work fully only with XeTEX and LuaTEX. With pdfTEX, the best fall-back is to cover just the ASCII range. Unlike the first part of this article, here we are also discussing code for LATEX3, thus at the expl3 programming level (The LATEX3 Project, 2015). The commands therefore have 'real' names that might seem unusual: to avoid obscuring the ideas, I'll give them 'design' (CamelCase) names in the examples here.

3.2 The approach

The first thing to recognise is that when we want to talk about case changing, we are talking about *text*. There may be some embedded formatting to skip (more on that in a bit), but we can work on the basis that we are case-changing category code 11 and 12 tokens. Of course the TEX primitives have important uses in generating 'funny' tokens (as they change character codes but not category codes): that's got essentially nothing to do with the case of characters at all!

With a bit of effort it's possible to set up an expandable loop over a list of tokens that preserves all of the spaces and brace groups. Using that approach, we can pull out tokens one by one for conversion. Spaces are passed straight through, while we need to use a recursive approach with groups. So this leaves the question of dealing with 'normal' tokens.

Converting each token is done by using a look-up table made up of 100 control sequences, each covering part of the full Unicode range. This approach offers a balance between efficiency and performance. Using a table of this form, we are not limited to one-to-one look-ups: one-to-many is also available. This core idea covers a large part of the situations we need, but to get the context dependence needs some specialised code. In the current approach, that is done using look-ahead routines dedicated to each special situation.

Covering language-dependent mappings needs a version of the code that tracks the currently-active language. The number of special cases we find for this is pretty small, so each one can then be handled using some custom code.

3.3 Features

The key features of the case changer are those related to the Unicode standards: the one-to-one mappings and more complex relationships follow those given by the consortium. The basics are built-in: detection of context and providing a way to indicate the language of the text as an additional argument to case changing. Thus

```
\edef\test{%
  \ExplLowerCase{RAGIP HULÛSİ}%
}
\show\test
```

yields

```
> \test=macro:
->ragip hulûsi̇.
```

whilst

```
\edef\test{%
  \ExplLowerCase[tr]{RAGIP HULÛSİ}%
}
\show\test
```

yields

```
> \test=macro:
->ragıp hulûsi.
```

We can see that in the second case we get not only full mapping of the Unicode characters but also the difference in treatment of the dotted and dotless I.

From a programmer's point of view, it's convenient to be able to do case changing expandably. As we can see in the examples above, that's exactly what the code offers: the case changer can be used inside an `\edef`. That means we can easily store

the 'real' case changed text without having to jump through any TEX primitive hoops.

At the user level, some things should be skipped by case changing: math mode material and explicitly-marked input. Following the pattern set up by the textcase package (Carlisle, 2004), these are handled by detecting $ (etc.) and using a dedicated 'opt-out' command, respectively. Thus we obtain

```
\edef\test{%
  \ExplUpperCase
    {Some maths $y = mx + c$}%
}
\show\text
...
> \test=macro:
->SOME MATHS $y = mx + c$.
```

and

```
\edef\test{%
  \ExplUpperCase
    \NoChangeCase{FeFe}-hydrogenase}%
}
\show\text
...
> \test=macro:
->\NoChangeCase {FeFe}-HYDROGENASE.
```

One of the subtle features of Unicode case changing is what they call titlecasing. This is the process whereby the first character of a piece of text is made upper case, with the rest being lower case. The subtle part is that a few characters need special handling if they are first: these tend to be situations where the single glyph looks a bit like two letters. We've called this "mixed case": titlecasing in English at least implies some form of word-level processing.

```
\edef\test{%
  \ExplMixedCase{ǆ}%
}
\show\text
...
> \test=macro:
->ǅ.
```

Perhaps the best example of this behaviour is with the combination IJ in Dutch.

```
\edef\test{%
  \ExplMixedCase{ijsselmeer}%
}
\show\text
...
> \test=macro:
->Ijsselmeer.
\edef\test{%
  \ExplMixedCase[nl]{ijsselmeer}%
}
\show\text
...
> \test=macro:
->IJsselmeer.
```

The final area to consider is *case folding*. This looks very much like lower casing but it's not meant for text: it's a process for programmers. Case folding is a strictly one to one mapping with no context dependence. We've provided this using a simplified approach (no special tests), and based on yet another Unicode data file.

4 Conclusions

Using Unicode data in TEX needs a bit of thought to match up the ideas of the two systems. However, we can do that and benefit from the tremendous amount of work done by the Unicode Consortium. In return, we enable users to get predictable outcomes from their code and to match up with the handling of other computational systems. This will only become more important in the future.

Thanks to Jonathan Kew for creating the Perl script used as a basis for our Unicode data parser. Thanks also to Bruno Le Floch who developed the looping approach used for case changing and the method for compacting the data into an efficient format.

References

Carlisle, David. "The textcase package". Available on CTAN: macros/latex/contrib/textcase, 2004.

Kew, Jonathan. "XeTeX". http://xetex.sourceforge.net/, 2015.

The LaTeX3 Project. "The expl3 package". Available on CTAN: macros/latex/contrib/l3kernel, 2015.

The Unicode Consortium. "The Unicode Standard". http://www.unicode.org/versions/latest/, 2015.

◇ Joseph Wright
Morning Star
2, Dowthorpe End
Earls Barton
Northampton NN6 0NH
United Kingdom
joseph.wright (at)
morningstar2.co.uk