Storing Unicode data in TEX engines

Joseph Wright, LATEX Project Team

1 Introduction

Unicode has become established over the past three decades as *the* international standard for representing text in computer systems. By far the most common input encoding in use today is UTF-8, in which Unicode text is represented by a variable number of bytes: between one and four. Unicode deals with *codepoints*: a numerical representation for each character. There are in principle 1 114 112 codepoints available, although not all are currently assigned and some of these are reserved for 'private use' for *ad hoc* requirements.

Each codepoint has many different properties. For example, depending on our application, we might need to know whether a codepoint is a (lower case) letter, how it should be treated at a line break, how its width is treated (for East Asian characters), *etc.* Unicode provides a range of data files which tabulate this information. These files are human-readable and are, in the main, purely ASCII text: they are therefore not tied to any particular programming language for usage. The full set of files is available from `unicode.org/Public/UCD/latest/ucd/`: the complete current set as a zip is around 6.7 MiB.

There are of course standard libraries for common programming languages such as C which both load this data and provide implementations of the algorithms which use this data: things like changing case, breaking text into lines and so on. However, these are not readily available to us as TEX programmers. Thus, if we want to be able to properly implement Unicode algorithms, we will need to look at how to load the relevant data and store it within TEX in an efficient manner.

Here, I will focus on how the LATEX team is approaching the data storage challenge. I will show how the particular requirements of implementing in TEX mean we need to use a mix of approaches, depending on exactly which data we are looking at. The current implementation for loading this data in `expl3` is available at `github.com/latex3/latex3/blob/main/l3kernel/l3unicode.dtx`, and is read as part of the LATEX 2_ε format-building process.

2 The data challenge

With over a million codepoints, even on a modern computer, storing values for every codepoint separately is impractical, particularly when we worry about having multiple properties and needing to access any data value with equal ease.

In some cases, we do not need to record properties for every single Unicode codepoint. We will see that for example with grapheme breaking: most codepoints have the same property value here, so we can tackle data storage by looking just at exceptions. However, there are major problems there. First, there are plenty of properties where we *do* need to track information of most if not all codepoints. There are cases where we might look at using ranges of characters, but the downside to this can be that we get uneven speed of access: that's fine if all of our documents are written in ASCII, but not acceptable if we need to cover a range of input scripts in an even manner.

This is not something that applies only to TEX of course, and it's a problem that the notes from the Unicode Consortium themselves address. The recommended approach is to use what is called a *two-stage* table. This is a way of covering all of those 1 114 112 codepoints without needing to store separate values for every single one, and maintaining fast access for all codepoints. We will see both how that works and how to do it in TEX below.

3 TEX aspects

For classical TEX engines, we might think we don't need to cover all of Unicode: the engines are only 8-bit anyway. But we know that we can take that 8-bit input and treat it as codepoints: the LATEX `inputenc` package has done that for over 30 years. So even if we don't need to cover *all* of Unicode, we need to handle a subset, and it's not always easy to make this a clearly limited and non-expanding subset. So even for these engines, we likely need methods to store a full range of data.

The Unicode engines XETEX and LuaTEX present a different question. They do have tables for some Unicode data: `\uccode`, `\lccode`, `\catcode` and so on. But whilst we do need to set those up (so that they have the 'right' values for TEX operations), it turns out they don't offer enough flexibility to track everything needed. Also, if we want to be able to use code shared by different engines, we want to use approaches that work with pdfTEX anyway. For the LATEX team, that's the case, of course.

The experienced TEX programmer might at this stage well be worrying about where I'm thinking of putting all of this data. TEX is very limited in the data structures it provides: macros and some registers. The latter are simply too limited in number, even with the ε-TEX extensions. We can store quite a bit in macros, and rely on the hash table to get fast access, but even that isn't going to scale well for the amount of data we might want, at least without

doi.org/10.47397/tb/44-1/tb136wright-unidata

some extra tricks. But there is another, perhaps unexpected, data store we can use, one that will give us quite a bit of headroom: font dimensions.

It turns out that we can set almost as many `\fontdimen` values for a font as we want, but we need to know how many to create for any given font. We don't want to do that for real fonts, but we can load the same font at lots of sizes and use each size as, effectively, an integer array. All that's needed is to pick sizes that the user doesn't care about: we do that by starting at 1 sp and working upwards. There is a limit on the *total* number of `\fontdimen` values, but it's in the millions and we won't get close to that. So we do have a fast random access data structure we can use for storing integer values. Now all we need to do is use this idea efficiently.[1]

4 Making it numerical

Before we deal with storing all of the Unicode data we need, there's the question of exactly what we will store. Very few Unicode properties are numerical: they are descriptors of behaviour. However, we can turn most of them into something we can represent by a number. The Unicode 'General Category' property is a good demonstration here. There are 31 possible values, for example

Cc	Control character
Lu	Uppercase letter
Nd	Decimal number

It's trivial to assign a numerical value to each of these; then we can store that integer value and quickly convert to the descriptor as required.

That approach works for most properties, but not, for example, for case-changing data. The case mapping of a codepoint will itself be a codepoint: it could be the same one, or it could be *anywhere in the Unicode range*. We are going to want values that have some chance of repeating, so storing the *absolute* value of the target codepoint isn't going to work. Instead, we will store the *relative* position of the 'output' codepoint. For example, `A` is `"0041` and `a` is codepoint `"0061`. So we will store the lowercase mapping for `A` as `"0041 - "0061`, *i.e.* −32. The open-ended nature of the values here is going to impose a few extra conditions, as we'll see in a bit.

[1] If we are working in LuaTEX, other data structures are available for storage. In expl3, the same macro-level interface is used for creating integer arrays in all engines, with LuaTEX using a Lua-based storage method. This allows an engine-neutral approach to the problem of storing large amounts of numerical data whilst still taking advantage of the greater flexibility of Lua where available.

5 Two-stage tables ...

The idea of a two-stage table is that it offers fast data access to a large number of values, while at the same time avoiding storing every single entry separately. This works for us here as there are patterns in the data we can exploit. Two-stage tables are recommended by the Unicode Consortium and are used by several languages. A particularly clear explanation, including an implementation for storing general category data written in Python, is available at `strchr.com/multi-stage_tables`.

The two-stage approach is based on arbitrary data blocks (not Unicode's character blocks): we divide the full Unicode range into equal-sized blocks, then deal with each block separately. The size of the block (a power of two) somewhat affects the amount of compression we will see, but anything from 64 to 256 gives similar results; these are typical values.

Dividing the full range into blocks of known size means of course that we know how many blocks there will be. For example, if we assume a block size of $n = 256$, there will be 4352 blocks. That will be the size of the first table we will use, with one entry for each of these blocks: that means it has a predictable size, and can be created before we do any data processing. Each entry in this first table points to a second table, of which we will need several.

The second stage tables contain the data for each block, so have n entries each. What we don't know here before creating the entire data structure is how many of these second stage tables we will need. At the start of building the structure, each block of codepoints will need a separate second stage table. But as we go on, we will find that different blocks can reuse the same second stage table. So, representing the table as a comma-separated list, we might see our first stage table (the property values we need to store) looking something like[2]

```
1, 2, 3, 1, 4, 5, 6, 1, 2, 8, 1,
...
```

That is, the first three values in the first-stage table each point to different blocks in the second stage table, but the fourth value points to the same second stage block as the first, and so on. As we get into the parts of Unicode that have long ranges of codepoints with identical property values, this compression effect becomes significant and the total size of the two stages ends up much smaller than the total number of codepoints.

[2] Here, I am using an index from 1: this is the approach used by expl3 and by Lua. Languages involving direct memory management will use an *offset* starting from 0.

With this all set up, retrieving a value is quite quick. We can find which block a codepoint is in, and the position within a block, with a couple of numerical expressions. So getting a value out of the tables is very fast. That of course is the point: the work is done in the creation stage, so at point of use everything is very quick.

To set this up in TEX, we need to think about exactly how to create those two tables. As I've said, we can predict the size of the first table, so we can make that directly using the `\fontdimen` approach. We can't do that for the second stage as we don't know in advance how many entries we will need. Also, we want to be able to check each block's table against those we've already created. That's better done if the data are stored in macros: a series of comma lists work well. Macros are fine if we don't need to access the values randomly, and during the creation stage that's true. We can then use fast `\ifx` tests to check each block as we finish it: have we seen this block before? Once we've done all the blocks, we can then create the second `\fontdimen` table in one shot. (We could make lots of stage-two tables, but as they are of predictable size, we can store all the information in a single `\fontdimen` array using an offset to get the right block information.)

With over a million codepoints, one might be worried about how long reading every one of them will take. However, in most cases, large parts of the full range are compressed in the input. For example, `UnicodeData.txt` contains details of case mappings and general category. For many east Asian characters, these and other values are identical, so the file simply lists the first and last entries with similar values. So for these, we don't have to work through every codepoint: we just have to work out which second stage table they use, then add the right number of entries to the first stage.

With some carefully-coded for loops, we can read the entirety of `UnicodeData.txt` and save all of the upper- and lowercase data in a couple of seconds. That needs only four `\fontdimen` arrays, and the total number of entries is fewer than half of the number of codepoints that have case data.

6 ... or not

As you will have seen, whilst a two-stage table approach is efficient for covering the whole Unicode range, there is a limit to the degree of compression, as the first stage will always have a significant number of entries, even if we need very few second stage tables. At the same time, the approach relies on being able to read the data once, so it's not so good if we want to make *ad hoc* changes. It should come as no surprise, therefore, that dealing with one-off overrides is best done using other methods, for example storing as macros which can then be looked up using TEX's hash table.

The line between using a two-stage table approach and individual hash table entries (or other approaches) is fuzzy: one needs to make a judgement. But broadly, if we are looking at fewer than a couple of thousand codepoints, we are likely to avoid a two-stage approach. For example, storing case folding and titlecasing information is easier using a macro approach: both are essentially tightly focussed variants of standard case changing, and apply only to a relatively small number of codepoints.

Another area where two-stage tables are more tricky to use is where we need to store multiple values. This applies for example to normal form decomposition and to full lower/uppercasing data. We could do that by having combined values in a first stage table, for example 1 to 999 for the first output codepoint and 1000 to 100 000 for the second. But the alternative of using a two-stage approach for the one-to-one data, then a hash approach for one-to-many, works pretty well for us.

Finally, there is a consideration about how we are actually loading the data. The source data file `UnicodeData.txt` is ordered by codepoint, so is ideal for reading line by line and turning the contents into a few two-stage tables covering the different concepts. Here are a few lines from `UnicodeData.txt`:

```
0000;<control>;Cc;0;BN;;;;;N;NULL;;;;
...
0041;LATIN CAPITAL LETTER A;Lu;0;L;;;;;N;;;;0061;
...
0061;LATIN SMALL LETTER A;Ll;0;L;;;;;N;;;0041;;0041
...
10FFFD;<Plane 16 Private Use, Last>;Co;0;L;;;;;N;;;;;
```

Several of the other Unicode data files are ordered for logical access. For example, the grapheme-breaking data file (`GraphemeBreakProperty.txt`) is divided up by breaking class, then within that ordered by codepoint. Some example lines, from three different classes (in the real file, the comments are not on separate lines):

```
0600..0605    ; Prepend
 # Cf   [6] ARABIC NUMBER SIGN..ARABIC NUMBER MARK ABOVE
...
00AD          ; Control
 # Cf       SOFT HYPHEN
...
AC01..AC1B    ; LVT
 # Lo  [27] HANGUL SYLLABLE GAG..HANGUL SYLLABLE GAH
...
D789..D7A3    ; LVT
 # Lo  [27] HANGUL SYLLABLE HIG..HANGUL SYLLABLE HIH
```

To turn that into a two-stage table, we first need to do some manipulation to get it into the right form.

We can do that, of course, but there's a time cost, and we would also have to worry about how many intermediate data structures we are using.

Many languages handle this problem using a dedicated script to make their two-stage table structures, then reading some 'digested' form back at runtime. For TEX use, that would probably be better done using a different scripting language: Python has some advantages, but as Lua is the TEX world's standard scripting system, I would favour that. The downside to this approach is you can't use the files from the Unicode Consortium directly, so you have to keep track of your digested set. Also, as in TEX we tend to create formats, and they already *are* digested data dumps, it feels more natural to just read the 'raw' Unicode files as part of format-building, wherever possible.

The outcome of that decision is that there are places where it's easier *not* to use a two-stage table, as we can make a reasonably efficient structure in macros that works 'well enough'. I've done that, for example, for grapheme breaking. There are only about 12 different grapheme breaking classes, and almost all codepoints are in the default one that we *don't* need to record. The raw data are ordered by breaking class, so it's easy to turn that into comma-separated lists of codepoint ranges: one list for each breaking class. Whilst this means that a few codepoints perform slightly less well than others when looking them up,[3] that's acceptable as most of the effort TEX is making here is not the data checking.

7 Outlook

Unicode is *the* way that most computer systems work with text data today. Supporting Unicode methods is workable in TEX, even with engines that are fundamentally 8-bit. Over time, more Unicode data will be needed by expl3, and potentially by others, and using the approaches outlined here we can make that available inside TEX runs without needing to look to novel engine extensions.

It's possible that as more data are required, it will be sensible to move from parsing in TEX to parsing in Lua for 'digestion'. But the underlying data structures can remain the same; that is only a question of how best to create them.

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

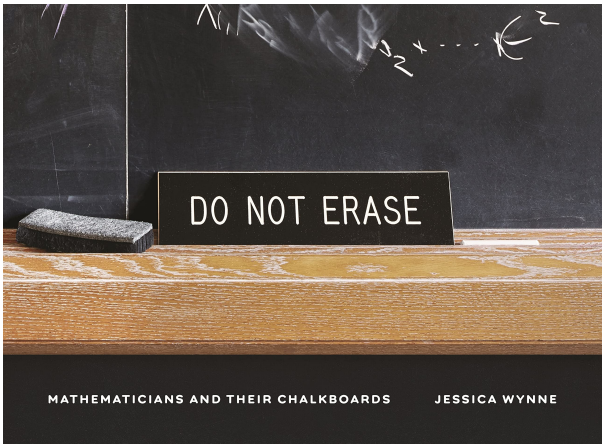
◇ LATEX Project Team
https://latex-project.org

[3] Hangul characters: these have the most complex breaking behaviour.

Book review: *Do Not Erase: Mathematicians and Their Chalkboards*, by Jessica Wynne

Jim Hefferon

Jessica Wynne, *Do Not Erase: Mathematicians and Their Chalkboards*. Princeton University Press, USA, 2021, 240 pp., hardcover, US$35, ISBN 9780691199221.



This book exhibits an art that is small but that is very important to many of us: the handwritten work of mathematicians.

The author introduces the project by describing some friends, "Amie and Benson are theoretical or 'pure' mathematicians ... One day on the Cape, I watch Benson work at his dining room table. ... For several hours, he sits, thinking, creating, jotting down the occasional note. It looks like he has a secret. ... I feel like he is creating something so expansive and beautiful it is beyond words, something that exists only in his head. When I ask him to explain what he is working on, he pauses, appears to struggle for the right words, and replies, simply, 'No. I can't.'"

Later the author also says, "I have always used my camera as a way to understand and explore the world." So while this book is subtitled *Mathematicians and their chalkboards*, another good way to understand it is: glimpses into the works of mathematicians through their chalkboards.

This is a coffee table book. It is beautiful, with typography that is understated yet powerful. It is not coffee table-sized but it has the same feel in that you shouldn't read it, you should browse it. Open it anywhere and you see a two-page spread about one mathematician. Most are research mathematicians, although a few do not fit that description perfectly. Even-numbered pages have a brief biography and an essay by that mathematician reflecting on their work. They write about their perception of beauty, or perhaps a bit about their career and what drew

doi.org/10.47397/tb/44-1/tb136reviews-wynne

them in the direction that they went. It is not the usual thing for mathematicians to put in writing.

Odd-numbered pages show that person's chalkboard. Each picture is different than the others and all are visually interesting.

These are not necessarily candid shots. The author says, "I ask the mathematicians to write or draw whatever they want on their boards. (Often I end up shooting whatever is already on the board — usually something they are currently working on.)" So the subject had the option to put on the board what they want to share. That's wise, if only because for instance my board would contain a grocery list, along with some passwords.

Some boards are messy, some are spare. Some are covered with formulas, while others focus on a figure or two.

I'll take Gilbert Strang as an example. There is a five-sentence biography and his essay focuses on what he is most popularly famous for, his lectures and book on Linear Algebra. His blackboard is taken straight from that class, with matrices and matrix equations hard at work.

The author asserts, "Despite technological advances (such as the creation of computers), chalk on a board is still how most mathematicians choose to work. As musicians fall in love with their instruments, mathematicians fall in love with their boards — the shape, the texture, the quality of the special Japanese Hagoromo chalk." While I don't know that this is completely true, since plenty of people prefer whiteboards, or paper, or tablets, perhaps it doesn't matter. Certainly chalk gives the pictures a theme. Certainly also many of us agree with Sun-Yung Alice Chang who says, "Despite the computer age we live in, the type of talks I enjoy the most are still those in which the speaker writes on the blackboard line by line and explains his or her thoughts."

There are many books about mathematics filled with beautiful graphics. Searching for phrases such as 'mathematics art' or 'mathematics beauty' will produce a list. They often have lots of drawings done with computers, such as fractals, and these can be stunning as well as fascinating. However, as with really high-end natural science illustrations, often somehow the hand-crafted ones are more compelling and show better what it is that the viewer needs to see.

This book also fits with another tradition, ones that reflect on a life in mathematics. Many of these are biographies but there are some that like this one touch on a number of mathematicians, sampling broadly rather than deeply. Two familiar and excellent ones of this type are *Mathematical People* and *More Mathematical People*. One that is recent enough that some readers may not yet have seen it is *Mathematicians: An Outer View of the Inner World*, published by the AMS in 2018.

Alec Wilkinson's afterword is a powerful meditation, "These photographs ... typify the mathematician's historic engagement with beauty." Strictly speaking this book doesn't have to do with TEX — for instance, there is no mathematical typography in the typeset material — but it is about conveying mathematics and about beauty. He closes by saying, "Each of these elegant photographs preserves a detail in the canvas of rigorous human thought."

Do Not Erase is an excellent choice as a gift for a budding mathematician, to communicate what the life of a mathematician is like, especially that of a researcher. For the same reason it is also a great choice for either an institutional or departmental library. The striking beauty draws the reader in and the essays hold them.

◇ Jim Hefferon
jhefferon (at) smcvt dot edu