Tsukurimashou: A Japanese-language font meta-family

Matthew Skala

Abstract

METAFONT-based font projects for the Chinese, Japanese, and Korean (CJK) languages have been announced every few years since the early 1980s, even predating the current form of the METAFONT language. Except for a few non-parameterized conversions of fonts that originated in other formats, in 30 years every METAFONT CJK font has been abandoned at or before the 8-bit barrier of 256 *kanji*, nowhere near the thousands required for practical typesetting. In this presentation I describe the first project to break that barrier: Tsukurimashou (`http://tsukurimashou.sourceforge.jp/`), currently at over 1500 *kanji* (as well as kana, Latin, and Korean hangul) and steadily growing. I discuss technical and human challenges facing this kind of project, how to solve them, and spin-off technologies such as the IDSgrep *kanji* structural query system.

1 Introduction

The Han script, used by the Chinese, Japanese, and Korean (CJK) languages among others, includes very many characters. Just counting them is tricky, but a human being might typically need to know a few thousand for basic literacy in a Han-script language. The list of 2136 characters taught in the Japanese school system (the *jouyou kanji*) is one benchmark, near the low end. Chinese requires more, and a typesetting system may require more still, because of rare characters found in names, historical contexts, and so on. A human being can get away with failing to read the occasional character; typesetting systems need to be able to print nearly all of them. Computer fonts considered usable for Japanese typically cover between six and twelve thousand characters. Databases of rare characters used in linguistic research cover tens or hundreds of thousands.

The sheer number of characters that go into a CJK font, and the amount of work implied by that number, is daunting. Considering the difficulty of building even a simple Latin font with METAFONT, it may be no surprise that there are no complete METAFONT-native CJK typefaces. But on the other hand, examination of Han-script text (even, or especially, by someone who cannot read it) quickly reveals that characters can be decomposed into smaller parts, as shown in Figure 1. Computer scientists who examine Figure 1 are likely to believe they understand it. “Of course,” one supposes, “the tens of thousands of Han characters are just a small vocabulary of primitive shapes, perhaps only a few dozen, which combine in straightforward ways according to a spatial grammar to form tree structures!”

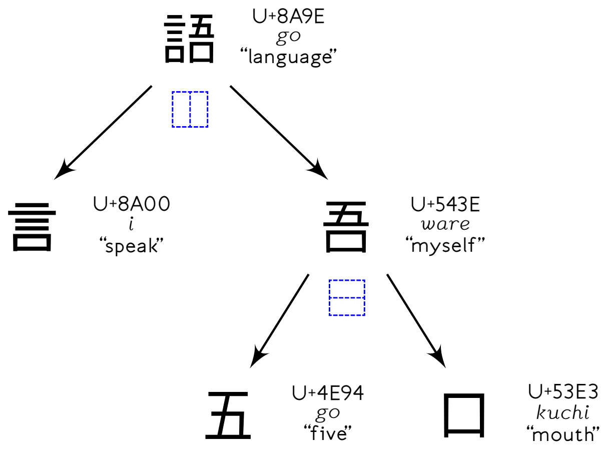

Figure 1: Breaking a character into its parts.

Computer scientists know how to deal with such things. It should be only the work of a week or two for a good programmer to lash together a prototype CJK font generator. Each primitive shape can be a subroutine; there can be other subroutines expressing the combining operations such as “place this one above that one”; a few parameters applied to the low-level shapes can allow for creating a wide range of styles; and the only real challenge is looking in the dictionary that lists the tree decompositions of all the characters. That book must exist in China, so we’ll get it by interlibrary loan. This project might even be easier than building a Latin font meta-family.

The earliest METAFONT CJK project I know of was LCCD, the Language for Chinese Character Design, described in a 1980 Stanford technical report by Tung Yun Mei [11]. Mei collaborated with Knuth and based LCCD on the METAFONT79 language developed to that point. Even in 1980, many of the ideas were already in place that a present-day computer scientist would naturally think of on viewing Figure 1. Mei’s report includes images of 346 “basic strokes and radicals”, and 112 completed characters.

Subsequent work on METAFONT-native CJK fonts includes that of Hobby and Guoan in 1984, who created 128 characters [5]; Hosek in 1989, character count unknown but two are displayed in the *TUGboat* article [6]; Yiu and Wong in 2003, in a project that targeted on-demand creation of rare characters rather than a font as such [16]; and Laguna circa 2005, with 130 characters in the last available version [10]. All these used a relatively small number of basic components, combining according to a spatial grammar to form more complicated characters.

I listed published METAFONT-related projects. Similar ideas have also been used behind closed doors in commercial font foundries (CDL from Wenlin Institute seems to be an example [15]), and non-METAFONT research projects like the LISP-based Wadalab toolkit [13]. The Wadalab font project ran during the 1990s; much of the work was lost or withdrawn, but some of its fonts survived to become widely used in the free software world. These kinds of projects use grammars of character parts, but lack the full parameterization that METAFONT users expect. There has also been work on using CJK fonts from other sources in TEX documents, sometimes including METAFONT incidentally in the workflow, but again without parameterization. For instance, the Poor Man's Chinese and Japanese package [12] converts bitmap fonts into METAFONT code that renders scaled versions (without smoothing!) at arbitrary resolution.

It may be difficult to create fonts in METAFONT in general, regardless of the script; but human beings have done it. Several, though not many, METAFONT-native Latin fonts exist, and we can typeset a wide range of documents in Latin-script languages with parameterized METAFONT-native fonts. So after more than three decades of work, why are there no usable, parameterized, METAFONT-native CJK fonts at all?

2 Scaling issues

It is no coincidence that the past attempts to build CJK fonts in METAFONT have been abandoned at the same stage in development, around 120 characters. *That is the roughly the size of a Latin font.* METAFONT was designed to build fonts with sizes on that order, and thus METAFONT users have built expertise and developed tools for building fonts the size of Latin fonts. When fonts get larger, unforeseen difficulties show up like *nurikabe* — the plaster wall monsters of Japanese folklore blamed for delaying travellers by night.

2.1 Technical limitations

Many font file formats are limited to 256 glyphs by their use of 8-bit character codes. People who attempt to typeset CJK documents in classical TEX use elaborate workarounds involving slicing their fonts into 256-glyph sub-fonts. Handling the input encoding for documents written in large character sets with these slicing schemes is a tough problem too, but fortunately not one we as font designers must solve. There are extended versions of the TEX interpreter designed to use longer character codes directly (XƎTEX is one), and those may also be able to work with font formats that store tens of thousands of glyphs per file and don't need to be sliced; but there is no similarly extended METAFONT to produce fonts in such formats.

Thousands of glyphs in a font does not just mean a bigger file. It also means more time spent compiling, and more memory consumption. One run of METAFONT may run out of memory or other resources trying to process an entire multi-thousand-glyph CJK font, and the user may run out of patience recompiling the whole thing after changing one glyph. To succeed at the thousand-glyph level, a project must have build tools allowing separate compilation of parts of the project. There should be tracking of dependencies among the different parts. Just being able to *find* pieces of code in a project this size — answering questions like "what was the name of the subroutine for such and such a shape?" — is an issue. These are elementary problems in software engineering, but there is little or no previous work on them in the METAFONT context because nobody has built systems this size in METAFONT before.

Classical METAFONT is designed to produce bitmap fonts, but bitmap fonts are no longer such a desired commodity. A present-day CJK font project will presumably target a vector format, but making METAFONT or some variation of it produce vector fonts requires additional layers of software, all of which are to some extent experimental. Bugs in the beyond-METAFONT software, previously undetected because previous fonts were smaller, will show up and need to be fixed. Keeping a handle on the bugs requires a test suite. The need for multiple steps in font compilation underscores the need for a capable build system. Human designers cannot be expected to issue five or six different commands in the right order to recompile every font, every time.

Earlier work on METAFONT CJK fonts has concentrated on writing code in METAFONT to draw the shapes of Han characters, as if that were the only problem to solve. Infrastructure that can scale to the size of the finished product is at least as significant.

2.2 Human factors

It is easy to underestimate how much work is involved in building a CJK font. We know how much work it is to design a Latin font. We know a CJK font has about 30 times as many glyphs. But it is easy to think, looking at Figure 1, that the CJK font should only be something like two or three times as much work as the Latin font (perhaps less), because so much code can be reused. In fact, less work is saved by code reuse than one might hope: every glyph requires some human attention. In computer science terms, font design is not much less than $\Omega(n)$.

Once it becomes clear that a human must spend time on every single glyph — it gets easier as more code exists to reuse, but there is no break point after which hundreds of characters will suddenly come for free — it is natural to hope that that human not be oneself. If we can just build a sufficiently good, easy to use set of tools, we can put them on the Web, maybe use a Wiki, and have many people in the community build a few glyphs each. Many hands make light work, once the infrastructure exists.

But to hope for someone else to build the actual glyphs after the tools are designed is to ignore a principal reason why people participate in free software projects in the first place. Designing tools for glyph construction is *fun*. Going through a list of 6000 glyphs one by one, doing simple repetitive tasks on each of them, is *work*. It is not easy to get volunteers for that sort of thing at the best of times, let alone when the volunteers must also have proficiency in an obscure programming language. The most successful large-scale collaboration is probably GlyphWiki [9], which sacrifices parameterization for a more purely graphical approach that demands less from the participants.

Finally, many of the potential rewards of a METAFONT CJK project, such as academic publications, can be had at the start, before the boring part; and then there are no more rewards until the end, and few then. You can publish one paper about your innovative techniques for building fonts; and you can publish one paper saying you have finished, years later. There is little in between. Knowing that this is the reward structure makes it tempting to write only the first paper and then start work on something else.

2.3 The script itself

The Han script itself may be the most ferocious *nurikabe*. Figure 1 with its clean decomposition of “language” into “speak”, “five”, and “mouth”, is deceptive. Many characters can be described as simply as that, but many others cannot. Consider Figures 2, 3, 4, and I could draw many more.

In Figure 2, “forest” is two copies of “tree” placed side by side. But the “tree” on the left is different from the “tree” on the right. If you make the two sides of “forest” look identical, readers will still know that you meant to write “forest”, but it will not look right. For a high-quality font, it has got to look right. This entails either creating two different primitives for the two trees, or having a smarter tree that knows how to change itself when it is on the left. Many character components change when they appear on the left. The modifications made when a component appears on the left are partially systematic, so we might hope to write code that can derive the left side shape automatically from the other shape, but it will not be simple, it will require manual supervision, and some projects have not gotten as far as noticing that it was an issue in the first place.

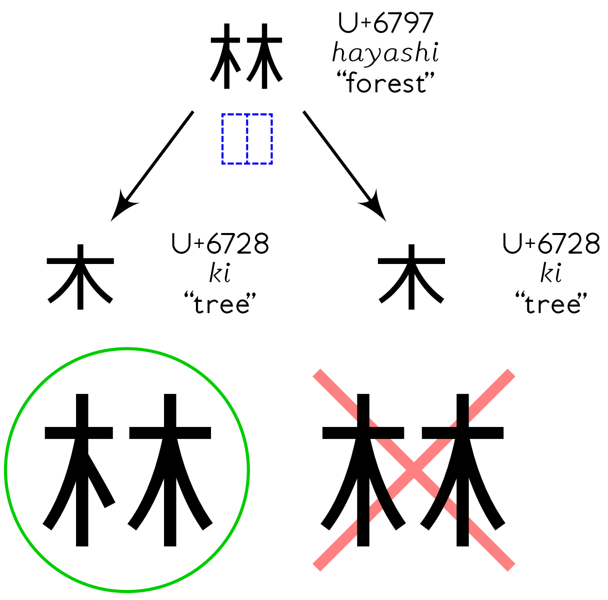

Figure 2: A forest is not two identical trees.

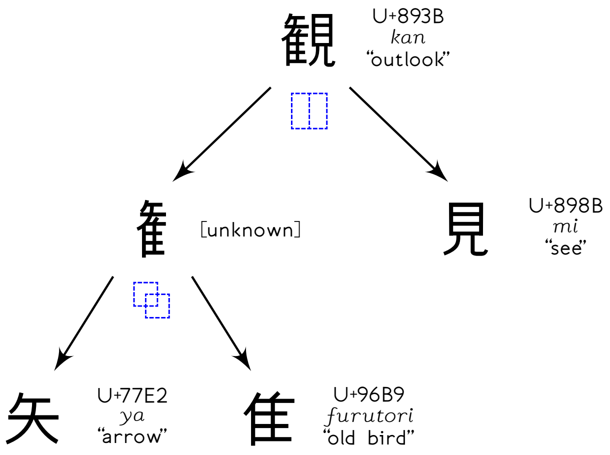

Figure 3: Combining operations can be complex.

In Figure 3, the left side of “outlook”, in addition to not being a character in its own right, is some kind of hard to describe combination of “arrow” and “old bird”. It is not good enough to just print a scaled copy of “arrow” on top of “old bird” and hope for the best; getting it right requires modifying and deleting strokes in both parts. A generic overlap operation is unlikely to be flexible enough to do the right thing here. Every character that contains this sort of thing requires specific human attention to adjust it beyond

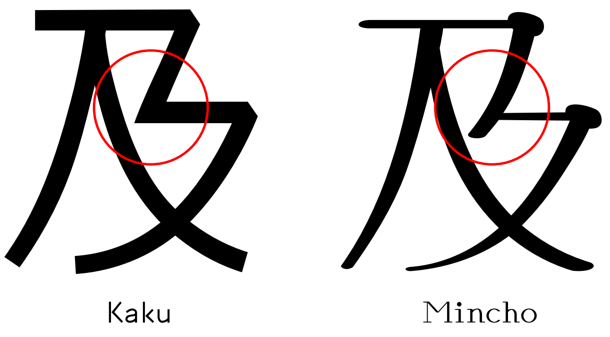

Figure 4: Two styles of U+53CA (*oyo*, "reach").

just saying "overlap". If the components change parametrically, then making sure they look right for all parameter values becomes even more complicated.

In Figure 4, two different styles of the same character are topologically different: the one on the left contains a single zigzag stroke that in the right-hand version is made up of two separate pieces. It is not easy to parameterize that in a way that will look good at every step in between, and if we make it a binary choice, giving up on the idea of interpolation, this difference will require some sort of "if" statement in the character description. A straightforward implementation of the grammar of shapes and combining operations suggested by Figure 1 would not provide for "if" statements.

These issues in the Han writing system point to an important conclusion: a simple grammar of parts and combining operations is not enough for building parametric fonts, even though it may be a useful starting point. Many characters can be decomposed into parts in the clean way implied by Figure 1, and such decompositions may be enough to support dictionary searches. It is easy to find enough well-behaved characters to put together a slide show or grant application, and to fool others or even oneself into thinking the whole character set will be easy.

But in order to produce high-quality fonts with full parameterization, with all the characters needed to typeset real documents, we must be able to override the simple descriptions and combinations of parts in arbitrarily complicated ways — per character and depending non-linearly on the parameters. To work at full scale, the font description language must have the power of a general-purpose programming language.

3 Tsukurimashou

My own attempt at building a METAFONT CJK font family is called the Tsukurimashou Project. The name means "Let's make something!"; it is an *anime* reference. As of version 0.8, released 26 August 2013, Tsukurimashou covers 1502 Japanese *kanji* (Han script) characters including all those taught in Japanese schools through Grade Four, as well as essentially complete coverage of *kana* (Japanese phonetic script), Latin, *hangul* (Korean alphabetic script), punctuation, and some miscellaneous ornaments and graphical characters. This is the work of one person, on a hobby basis while doing other things full-time for pay, since late 2010. It remains far from being a complete font family usable for typesetting general documents in Japanese, but it is already far past the point reached by any previous parameterized METAFONT-native CJK font project, and I believe my project is the first with a credible prospect of eventually reaching complete coverage.

Here are some points of reference distinguishing Tsukurimashou from other projects already discussed:

- Tsukurimashou is a parameterized meta-family, not a single font or a collection of independent fonts.
- Tsukurimashou is a font project, not primarily a dictionary of characters.
- Tsukurimashou is code, not data.
- Tsukurimashou is intended to achieve full coverage, at least of the characters needed for basic literacy in Japanese; it is not a proof of concept.
- Tsukurimashou is one person's non-commercial project; not a for-profit corporate or large-scale collaborative effort.

Tsukurimashou is hosted as a free software project on SourceForge Japan, with the bilingual project home page at `http://tsukurimashou.sourceforge.jp/` featuring downloadable packages, a Subversion repository for the source code, a bug tracker, mailing list, and so on. The package as a whole is distributed under the GNU General Public License, version 3, with a clarifying paragraph added to explicitly permit embedding the fonts in documents.

3.1 Motivation

The issues of human labour described in the previous section make it difficult for a CJK METAFONT project to reach complete coverage. Tsukurimashou's solution to the amount of work involved in font design is to redefine that large amount of work as *the main goal* of the project instead of *an unfortunate cost* of the project. This point alone seems to be largely responsible for Tsukurimashou's success to date.

I want to learn to read Japanese. Learning to read entails spending some time practicing and studying every character. But just studying a book and tracing copies on paper, as well as being boring, is not a particularly effective way to learn. I would also like to become skilled at using METAFONT and related font technologies. I believe I acquire skills best by completing tasks that require the skills. Designing a font family for Japanese, as a project that requires knowledge of the *kanji* and of METAFONT, including concentration on every character in turn, is a good way to acquire that knowledge. And from that point of view, the actual finished fonts are not even important. The fonts are my excuse for spending time thinking about every character, which is the real goal. With that goal in mind, *avoiding* human attention to every character stops being necessary or even desirable.

Of course, the project may have desirable side effects. Work on Tsukurimashou has required me to invent new technology that may be useful in other projects. Some of it is publishable research in computer science, certainly welcome for someone hoping to establish an academic career. And because it places heavy (in some cases unprecedented) demands on other free software systems, Tsukurimashou has proven useful in the development of those systems. Given that I am already committing to spend some time per character on learning the language, the hope is to make that time pay off in as many ways as possible.

3.2 A brief tour of the fonts

Tsukurimashou as a software package generates OpenType font files as its main output. Those are intended for use in general typesetting and word processing, not only within the TEX world. I most often use them with XeTeX. The OpenType fonts are divided up into families, of which the main supported ones are named Tsukurimashou, TsuIta, and Jieubsida; then there is parameterization within each family for overall style, boldness, and monospace or proportional spacing. The main supported styles for the Tsukurimashou family are "Kaku" (a traditional sans-serif style), "Maru" (sans-serif with rounded stroke ends), "Mincho" (a less traditional version of the common Mincho serif style), and "Bokukko" (which somewhat resembles handwriting with a felt-tipped pen). Finer-grained parameters are used internally and could be made visible by modifying the code, much in the way that Computer Modern has internal parameters like "`stem_corr`" as well as preset styles like "Roman". Figure 5 shows a sample of the font styles; Figure 6 shows more of the Japanese characters in the Mincho style. Version 0.8 with all options enabled will build a total of 120 OpenType files, including some that are experimental and not intended for actual use.

These are outline fonts intended for printing at high resolution. They contain hinting for bitmap conversion, but it is done automatically and not expected to be extremely high quality. Japanese-language typesetting has traditionally used monospace metrics, simple scaling (i.e., no corrections for optical weight), and no slanting or italicization; Tsukurimashou currently offers a choice between monospace or proportional, no optical weight features, and italics for the Latin script only.

Tsukurimashou Font Meta-Family
さてさて、何が出来るかな？

Kaku 角	Extra Light 白字
Mincho 明朝	Light 軽字
Maru 丸	Normal 本
Bokukko 僕女	**Demibold 半太字**
Monospace	**Bold 太字**
Proportional	**Extra Bold 黒字**
TsuIta Atama PS	ツイタ頭 *PS*
TsuIta Soku PS	ツイタ足 *PS*
Jieubsida 지읍시다	Dodum 돋움
Batang 바탕	Sun-Moon 선문

Figure 5: A sample of the Tsukurimashou meta-family of fonts.

わらやまはなたさかあ　ワラヤマハナタサカア
ゐり　みひにちしきい　ヰリ　ミヒニチシキイ
　るゆむふぬつすくうん　ルユムフヌツスクウン
ゑれ　めへねてせけえ　ヱレ　メヘネテセケエ
をろよもほのとそこお　ヲロヨモホノトソコオ
　一七三上下中九二五人休先入八六円出力十千
　口右名四土夕大天女子字学小山川左年手文日
　早月木本村林校森正気水火犬玉王生田男町白
　百目石空立竹糸耳花草虫見貝赤足車金雨青音

Figure 6: *Kana* and Grade One *kanji* in Tsukurimashou Mincho.

Although the largest use of Tsukurimashou fonts to date has been for typesetting the project's own documentation in English, the design of the Tsukurimashou Latin glyphs, especially in the Mincho style, is intended primarily for setting the short fragments of English that sometimes occur in Japanese text. Tsukurimashou Mincho used for pure English text ends up looking like a display face and might not be appropriate for entire sentences and paragraphs. Tsukurimashou Kaku is more suitable for extended settings in English.

The Jieubsida[1] family is intended to support the Korean *hangul* (alphabetic) script. *Hanja* (the Korean equivalent of *kanji*) are not included. This character set is relatively orthogonal: the main sequence of 11172 glyphs is algorithmically generated from a few tens of basic parts, though many less common letters had to be defined with more human intervention. Work on these fonts has proven useful in debugging the infrastructure at full scale, given that the Tsukurimashou series of fonts will eventually grow to a significant fraction of the size already reached by the Jieubsida series.

Beyond the main Tsukurimashou package, there are several smaller software packages called “parasites”, which appear in subdirectories of the distribution or may be detached. Some of these are font packages that share some of the Tsukurimashou infrastructure without really being part of the same metafamily; others are related software of other kinds. The only one discussed here will be the IDSgrep structural query system.

3.3 The infrastructure

Tsukurimashou’s infrastructure is designed like a typical free software project. It has source code that compiles into binary files, it has build scripts to accomplish that, and a would-be user can download a tarball, unpack it, and type `./configure` and `make`.

The build system is based on GNU Autotools. Choosing which source code files are needed for which font styles involves doing some logical inference that would not be convenient to do in a Makefile, so the Makefiles invoke additional code written in a subset of Prolog to evaluate the style selections, then run Perl scripts that scan the METAFONT sources to look for dependencies. The results of that computation are written into additional Makefiles, which guide the actual compilation process.

Knuth’s METAFONT creates bitmap fonts, while Tsukurimashou’s target is OpenType outline fonts. There are several METAFONT variants that can produce outline output from METAFONT source. I chose MetaType 1 [7] for Tsukurimashou. This package originates with the Polish TeX users group GUST and may be most famous for its use in the Latin Modern project [8]. It consists primarily of a macro package for MetaPost and a postprocessing script for GNU `awk`. One run of MetaPost generates the glyphs of a font as EPS files; another generates metrics; then the `gawk` script merges those and does some rewriting of the PostScript code to turn them into a single PostScript Type 1 font.

In recent versions, Tsukurimashou’s version of MetaType 1 has diverged somewhat from the one distributed by GUST. I started with the (very old) `mtype13` distribution, tried to upgrade it to use the latest MetaType 1 scripts, and ended up rewriting large sections of code. Many features of MetaType 1 are not used in Tsukurimashou (for instance, hinting; the “metrics” pass; and the entire processing chain in the reverse direction from PostScript back to METAFONT), and it proved useful to remove them, streamlining the code considerably. The core flow of information through Tsukurimashou’s version of MetaType 1 remains similar to that of the original, however: the MetaPost interpreter executes code in the METAFONT language, writing one EPS file for each glyph, and then those are postprocessed into PostScript Type 1 fonts.

Each PostScript font contains up to 256 glyphs (but usually far fewer than that), corresponding to a 256-character block of the Unicode character space. Many of these PostScript fonts are needed for each full-coverage OpenType font. The build system runs each individually through a FontForge script that removes overlapping sections of splines, this being an easier operation in FontForge than METAFONT. Once all PostScript fonts for an OpenType font have had their overlaps removed, it runs another FontForge script to combine them into the final OpenType font. Doing the overlap removal as a separate step is an optimization for the common case during development where only some of the PostScript fonts have changed: it reduces the amount of work needed to reassemble the updated OpenType font.

There are additional stages of processing in FontForge after the PostScript fonts are merged. The raw outlines generated by METAFONT may contain excessive or poorly-located spline control points; scripts in FontForge attempt to remove those. Similarly, some technical rules of the font formats (such as having points at the x and y extrema of each curve) need to be enforced. There is another processing chain for automated horizontal spacing and kerning of the proportionally-spaced styles. In that chain, the build system generates bitmap fonts in BDF format and a C program calculates spacing corrections, which are then applied back to the merged OpenType fonts. Other scripts are run on the side to do things like constructing OpenType glyph-substitution tables for Korean *hangul* support, and collecting data for proof generation. According to recent statistics

[1] Intended as a translation to Korean of the name “Tsukurimashou”, but I am informed that “Mandeubsida” would be a better translation, and am considering changing it.

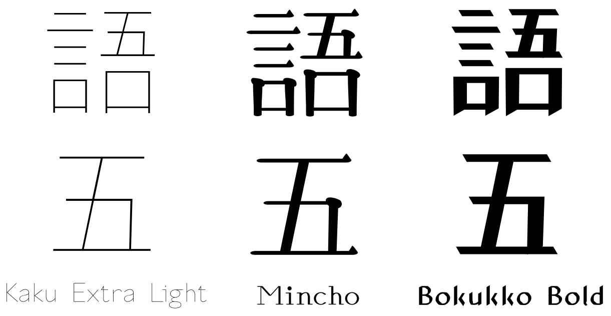

Figure 7: Three styles of "language" and "five".

from Ohloh [2], 63% of the project's code is written in MetaPost (the font descriptions proper), 8% is in LaTeX (documentation), and the remaining 29% is spread among 11 other programming languages: the infrastructure and some small spin-off packages.

3.4 The METAFONT code

Below is Tsukurimashou's code defining the "language" glyph of Figure 1; three styles of it are shown at the top of Figure 7. This glyph is of about average complexity; some are even simpler, and a few involve much more complicated operations, such as calculating positions of strokes based on the intersections of other strokes, or doing interpolation and conditional processing on style parameters.

```
vardef kanji.grtwo.language =
  push_pbox_toexpand(
    "kanji.grtwo.language");
  build_kanji.level(build_kanji.lr(450,0)
    (kanji.grtwo.word)
    (tsu_xform(identity yscaled 0.95)
      (kanji.grnine.my)));
  expand_pbox;
enddef;
```

This code is in a file named `tsuku-8a.mp`, which covers the Unicode code points U+8A00 to U+8AFF. A character like this one, which happens not to be used as part of any other character, is defined right there in the Unicode-range MetaPost file. Parts that are shared among more than one such file are moved to other files that can be included in multiple places; for instance, `kanji.grtwo.word` is in `gradetwo.mp`. Splitting macro definitions across many files like this makes it easier to avoid recompiling the whole system when something changes, but it also requires the build system to keep track of all the inter-file dependencies.

Tsukurimashou frequently uses a sort of functional programming via METAFONT's concept of text arguments to macros. A global stack data structure contains several kinds of objects to eventually be rendered into the glyph. A macro receives one or more arguments that are themselves fragments of code; it runs them, then examines the objects they added to the stack and possibly makes modifications. Macros that create *kanji* or parts of *kanji* normally put them into a square of arbitrary two-dimensional space defined by the coordinates from $(50, -50)$ to $(950, 850)$; the outer-level macros can then shift and scale that square into its final location in the finished glyph.

The macro `build_kanji.lr`, which combines things left-to-right, allows its two arguments to run, then scales and shifts their results to cover two smaller rectangles. The numeric arguments $(450, 0)$ specify that in this case, the dividing line is at x coordinate 450, and the two rectangles overlap by an amount of 0. So the left side runs from $(50, -50)$ to $(450, 850)$ and the right side is from $(450, -50)$ to $(950, 850)$.

Many of the visual adjustments needed when parts are combined, can be had just by choosing the right values for the dividing line and overlap amount. But other macros seen in this sample include `build_kanji.level`, which adjusts the stroke widths in its argument to all be the same (which often, but not always, looks better) and `tsu_xform`, which applies an additional METAFONT transformation matrix to make `kanji.grnine.my` a little smaller. Even in this relatively simple glyph, some tweaking was necessary beyond just putting together existing pieces in a standardized way.

Now, let's look at the code for the *kanji* numeral "five", which is invoked indirectly by `kanji.grtwo.language` when it calls `kanji.grnine.my`. This glyph is shown at the bottom of Figure 7. It is a typical example of the basic shapes that are not made up of smaller components.

```
vardef kanji.grone.five =
  push_pbox_toexpand("kanji.grone.five");
  push_stroke((170,740)--(830,740),
    (1.6,1.6)--(1.6,1.6));
  set_boserif(0,1,9);
  push_stroke((500,740)--(350,20),
    (1.6,1.6)--(1.6,1.6));
  push_stroke(
    (220,410)--(730,410)--(720,20),
    (1.5,1.5)--(1.5,1.5)--(1.4,1.4));
  set_boserif(0,1,4);
  set_botip(0,1,1);
  push_stroke((120,20)--(880,20),
    (1.6,1.6)--(1.6,1.6));
  set_boserif(0,1,9);
  expand_pbox;
enddef;
```

The `push_stroke` macros save paths on the stack, with each stroke defined by one path for the spine of the stroke, and a second path describing how the stroke weight (eventually translated to "width" through a style-dependent matrix) changes along the length of the stroke. Other macros, such as set_boserif, push other objects on the stack to indicate where serifs (*uroko*) should be added in styles that use them.

The whole thing, like kanji.grtwo.language before it, is bracketed by push_pbox_toexpand and expand_pbox, which respectively save, and adjust the size of, an object called a "proof box".

After all the macros that specify a glyph have run, rendering code unwinds the stack and generates outlines for all the objects, writing them to the PostScript output. This code is where most aspects of the font style are applied. Styles define the pens used for stroking, transformations for calculating pen size, the shape of serifs and whether to use them, and can potentially override parts of the rendering code by defining hook macros to apply further effects.

I have never fully understood METAFONT's traditional proof system based on greyscale fonts and "literate" programming, and in any case its reliance on the standard coordinate array z[] would not mix well with Tsukurimashou's object stack concept. Tsukurimashou generates proofs in a completely different way. When unwinding the stack the rendering code writes a "proof file", essentially a machine-readable log of all the things it is rendering. The build system collects the proof files and runs them through Perl scripts which generate TikZ/LaTeX files for an illustrated and cross-referenced edition of the source code. The proof boxes from push_pbox_toexpand result in annotations on the pictures, showing which part of each glyph came from which macro. Some information from the proof files also feeds into the kerning program, and is used for purposes like advising FontForge of white-on-black reversed glyphs, which represent exceptions to the overlap-removal rules otherwise applied.

4 Character databases and IDSgrep

Adding characters to Tsukurimashou requires knowing what is already in the system and what is in the language. For example, when looking at something like the left side of "outlook", I need to know whether such a thing already exists as a macro somewhere in the code base; whether many other characters in the language also include it (which would support the decision to create a new macro for future use); and which of its parts may be related to common shapes that could be used as guides for the new code. There are also simple coding questions to be answered, like "What was the name of that macro?" and "Which source code file is it in?"

More generally, anyone working with Han characters who does not read them fluently may wish to search a dictionary on partial descriptions: "What is this character I don't recognize that has 'speak' on the left and 'five' at the upper right?" Existing dictionaries sometimes offer what is called "multi-radical" search, whereby the user can specify one or more components and then see a list of all *kanji* that contain all those components. But multi-radical search features seldom if ever capture structural information like "on the left"; such a system would just show all the characters that contain "speak" in one pile for the user to dig through. In the initial stages of laying out Tsukurimashou's *kanji* support, I frequently found myself wishing I could use the power of Unix regular expressions, or something like them, to make more precise queries: why not run grep on the writing system itself?

The IDSgrep package attempts to serve that need. With some irony intended, IDSgrep's stated goal is to bring the user-friendliness of grep to Han character dictionaries. IDSgrep is one of the Tsukurimashou parasites: it comes included with the full distribution in a separate directory, or can be distributed on its own.

Recall the tree decomposition of Figure 1. That tree might be rendered into a simple ASCII-based prefix notation as "[lr](speak)[tb](five)(mouth)": it is a left-right combination of two things, the first of which is "speak" and the second is a top-bottom combination of "five" and "mouth". As argued earlier in this paper, such descriptions are not enough to render high-quality glyphs; but maybe if we include a few general catch-all categories like "overlap", and accept that not all descriptions will be detailed enough for rendering graphics, we can come up with a description for every character sufficient to offer useful dictionary searches.

The Unicode standard specifies syntax for Ideographic Description Sequences (IDSes), intended to support exactly this kind of pursuit [14]. There are special characters defined in the range U+2FF0 to U+2FFB to represent the prefix operators. Figure 8 shows some examples of the notation. Note the way the IDS notation conceals some details: for instance, the two sides of "forest" are both denoted by the same character, even though they look different when rendered. This looks promising: maybe we could get away with "just running grep" on a database of such decompositions.

In practice there are some additional challenges. For theoretical reasons, namely the difference between regular and context-free languages, a true regular expression search on these descriptions may

【林】⿰木木

【語】⿰言⿱五口

【観】⿰⿻矢隹⿱目儿

【涼】⿰水⿱亠⿱口小

【葉】⿱艹⿱世木

Figure 8: Unicode Ideographic Description Sequences.

be less than satisfactory. IDSgrep implements a tree-matching query language in which the user can specify character components to search for explicitly, or use matching operators like wildcard, match-anywhere, Boolean operations, and so on. The IDS syntax is not quite sufficiently flexible and well-defined to encompass all the tasks IDSgrep demands of it, and the special Unicode combining operation characters are difficult to type (and to typeset in Computer Modern!); so IDSgrep defines extensions to the syntax and ASCII synonyms for the special characters, forming a language of Extended Ideographic Description Sequences (EIDSes) that subsumes the Unicode IDS syntax.

IDSgrep's user interface is a Unix command-line utility similar to `grep`. It reads a database of trees in EIDS syntax, from files or standard input, and writes out any that match the matching pattern specified on the command line: just like `grep`. The syntax for matching patterns is complicated because it is powerful, but no worse for skilled users than standard regular expressions. After learning the syntax, a user can easily and quickly compose queries like "What characters have this component in that location, but not that other component anywhere?"

The latest version, IDSgrep 0.4, uses Bloom filters and binary decision diagrams to speed up searches. Although the full tree-matching algorithm is not slow, a complete search of hundreds of thousands of *kanji* dictionary entries may take a few seconds. So during installation, IDSgrep precomputes bit vector indices for the databases being installed; when searching those databases, it can do quick tests on the bit vectors to reject the large majority of possible matches, running the more expensive tree match on the candidates that make it past the bit vector check. The amount of speed-up is variable, but typically around a factor of 15.

But a critical question remains: where does the data come from? Databases of *kanji* marked up with structural data are not easy to find, let alone in IDSgrep's native format. The Tsukurimashou fonts generate (using information extracted from the proof files) a dictionary of character decompositions *as the characters appear in the fonts.* Querying how Tsukurimashou decomposes a character is often useful, but Tsukurimashou by definition does not cover the characters I have yet to add, and its decompositions may not reflect traditional etymology and other concerns. IDSgrep also ships with code to extract EIDS character decompositions from the KanjiVG Project's XML files [1] and from the CHISE IDS database [4]. It can do a "join" of any of the *kanji* databases with EDICT2 [3] to create an experimental dictionary of words and meanings with character decompositions. None of these databases is perfect; but especially by searching several at once, users can usually succeed in finding what they are looking for.

5 Conclusions and future work

There has been much past CJK METAFONT work, with few results and no finished fonts. I have described my own project, the Tsukurimashou parametric font meta-family, which is unfinished too. However, Tsukurimashou has made more progress than any similar system to date. I have described issues facing this kind of project, Tsukurimashou's solutions for some of them, and associated technology including the IDSgrep *kanji* structural query system.

The obvious direction for future work is to complete Tsukurimashou's *kanji* coverage. My hope, however, is that some of the code and ideas from this project will also be applicable in other languages and other projects.

References

[1] Ulrich Apel. KanjiVG. `http://kanjivg.tagaini.net/`.

[2] Black Duck Software. The Tsukurimashou open source project on Ohloh: Languages page. `https://www.ohloh.net/p/tsukurimashou/analyses/latest/languages_summary`.

[3] Jim Breen. The EDICT dictionary file. `http://www.csse.monash.edu.au/~jwb/edict.html`.

[4] CHISE Project. `http://www.chise.org/`.

[5] John D. Hobby and Gu Guoan. A Chinese meta-font. *TUGboat*, 5(2):119–136, November

1984. http://tug.org/TUGboat/05-2/tb10hobby.pdf.

[6] Don Hosek. Design of Oriental characters with METAFONT. *TUGboat*, 10(4):499–502, December 1989. http://tug.org/TUGboat/10-4/tb26hosek.pdf.

[7] Bogusław Jackowski, Janusz Nowacki, and Piotr Strzelczyk. Programming PostScript Type 1 fonts using MetaType1: Auditing, enhancing, creating. *TUGboat*, 24(3):575–581, 2003. http://tug.org/TUGboat/24-3/jackowski.pdf.

[8] Bogusław Jackowski and Janusz M. Nowacki. Latin Modern: Enhancing Computer Modern with accents, accents, accents. *TUGboat*, 24(1):64–74, 2003. http://tug.org/TUGboat/24-1/jackowski.pdf.

[9] Koichi Kamichi. GlyphWiki. http://en.glyphwiki.org/wiki/GlyphWiki:MainPage.

[10] Javier Rodríguez Laguna. Hóng-Zì: A Chinese METAFONT. *TUGboat*, 26(2):125–128, 2005. http://tug.org/TUGboat/26-2/laguna.pdf.

[11] Tung Yun Mei. LCCD, a language for Chinese character design. Report STAN-CS-80-824, Stanford University, Department of Computer Science, 1980. ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/80/824/CS-TR-80-824.pdf.

[12] Tom Ridgeway. Poor Man's Chinese and Japanese, 1990. http://www.ctan.org/tex-archive/fonts/poorman.

[13] Tetsuro Tanaka. Wadalab-Toolkit. Web page in Japanese. http://gps.tanaka.ecc.u-tokyo.ac.jp/wadalabfont/.

[14] Unicode Consortium. Ideographic description characters. In *The Unicode Standard, Version 6.2.0*, section 12.2. The Unicode Consortium, Mountain View, USA, 2012. http://www.unicode.org/versions/Unicode6.2.0/ch12.pdf.

[15] Wenlin Institute. Character description language. http://www.wenlin.com/cdl/.

[16] Candy L. K. Yiu and Wai Wong. Chinese character synthesis using MetaPost. *TUGboat*, 24(1):85–93, 2003. http://tug.org/TUGboat/24-1/yiu.pdf.

◇ Matthew Skala
Department of Computer Science
E2–445 EITC
University of Manitoba
Winnipeg MB R3T 2N2
Canada
mskala (at) ansuz dot sooke dot bc dot ca
http://ansuz.sooke.bc.ca/