作りましょう 0.6

パラメタほうしきフォントファミリ
ユーザマニュアル

Tsukurimashou 0.6

Parametric Font Family
User Manual

Matthew Skala
mskala@ansuz.sooke.bc.ca

2012年6月18日　　June 18, 2012

This project's English-language home page is at
http://en.sourceforge.jp/projects/tsukurimashou/.
このプロジェクトは、日本語のページが
http://tsukurimashou.sourceforge.jp/です。

User manual for Tsukurimashou

イントロ
Introduction

日本ごのユーザマニュアルをまちうけば、ごめんなさい。みらいは、日本ごのユーザマニュアルをかきます。

I want to learn Japanese. That's a large project, likely to involve years of daily study to memorize a few tens of thousands of words. It's clearly within the range of human instrumentality, because many millions of Japanese people have done it; but most of them started in infancy, and it's widely believed that people's brains change during childhood in such a way that it's much harder to learn languages if you start as an adult. I also would like to be fully literate within somewhat less than the 15 years or so that it takes a native learner to gain that skill.

What about designing a Japanese-language typeface family? Typeface design is a difficult activity, requiring hours of work by an expert to design each glyph. A typeface for English requires maybe 200 glyphs or so; one for Japanese requires thousands. That makes the English-language typeface about a year's full-time work, and the Japanese-language typeface enough work (20 to 25 man-years) that in practice it's rare for such a project to be completed by just one person working alone, at all. There is also the minor detail that I said a "typeface family"—one of those typically consists of five or six individual faces, so the time estimate increases to between 100 and 150 years.

However, I've already decided to spend the time on the language-learning project. And I'm sure that if I design a glyph for a character, I'm going to remember that character a lot better than if I just see it a few times on a flash card. And as a highly proficient user of computer automation, I have access to a number of efficiency-increasing techniques that most font designers don't. So the deal is, I think if I study the language *and also design a typeface family for it*, I might be able to finish both big projects with less, or not much more, effort than just completing the one big project of learning the language alone. And then I'd also end up with a custom-made Japanese-language typeface family, which would be a neat thing to have. And so, here I am, building one.

Because this is a parametrized design built with Metafont, users can generate a potentially unlimited number of typeface designs from the source code; but I've provided parameters for several ready-made faces, each of which can be drawn in monospace or proportional forms. The mainline Tsukurimashou fonts are intended for Japanese, but version 0.5 also introduced a series of fonts called Jieubsida, designed for Korean hangul. There is also some experimental code intended to eventually become a set of blackletter fonts, but it is disabled by default and not currently usable.

The main goal of this project is for me to learn the kanji by designing glyphs for them, and so far it does seems to be helping my learning process, though the nature of computer programming is certainly having some unusual disruptive effects. For instance, in some cases I appear to have unintentionally memorized the Unicode code points of kanji without learning their meanings or pronunciations. I may end up learning to speak Japanese just like a robot; but by the time I'm fluent, the Japanese demographic collapse will be in full swing and they'll have replaced much of their population with humanoid interfaces anyway, so maybe I'll fit right in.

Please understand that the finished product is not the point so much as the process of creation, hence the name. Furthermore, although the fonts cover the MES-1 subset of Unicode, and thus can in principle be used for almost all popular languages that use the Latin script, and I know that at this stage most users are probably more likely to use the fonts for English than for anything else, nonetheless these fonts are intended for eventual primary use with the Japanese language. Some decisions on the Latin characters were driven by that consideration—in particular, the simplistic serif design and weighting in the Latin characters of Tsukurimashou Mincho, and the limited customization of things like diacritical mark positioning in Latin characters not used by English. I biased some marks (notably ogonek and haček) toward the styles appropriate for Czech and Polish as a nod to my own ancestors, but I cannot read those languages myself, and I cannot claim that these fonts will really look right for them. I'm not interested in spending a lot of time tweaking the Latin to be perfect because it's not really the point. I already know how to

read and write English.

Some other notes:

- Since my learning the kanji is a big part of the goal of this project, “labour-saving” approaches that would relieve me of having to look at all the kanji individually myself (for instance, by feeding a pre-existing database of kanji shapes into my existing general font technology) are not really of interest.

- Proportional spacing and kerning still require some work. Be aware that future versions will change the spacing of some characters, so if you are one of those people to whom any changes in line breaking are anathema, you should not expect to be able to upgrade the proportionally spaced versions of these fonts in archived documents. The monospace versions have more long-term stability.

- I would like to include at least some support for vertical script, but it is not a high priority. One obstacle is that I don't have access to competent vertical typesetting software, whether the font could support it or not.

- Tsukurimashou is designed primarily for typesetting Japanese, secondarily for English. I have no immediate plans to support other Han-script languages (such as any dialect of Chinese) nor put a lot of effort into tweaking the fine details of characters only intended for use in occasional foreign words.

- Support of Korean is limited because of my limited knowledge of that language; and learning it is not a high priority for me. At this point the Jieubsida fonts only support Korean hangul, not hanja (which are the Korean equivalent of kanji, but just different enough that copying over the Tsukurimashou kanji would not be good enough).

- I reserve the right to add features that I think are fun, even if they are not useful.

- Tsukurimashou is designed as a vector font, assuming an output device with sufficient resolution to reproduce it. In practice, that probably means a high-quality laser printer. I have not spent time optimizing it for screens or low-resolution printers, and the hinting is automated.

- If it turns out to be too much work after all, I might abandon the whole project.

- Both building and use of Tsukurimashou require working around many bugs in third-party packages, some of which were mentioned here in earlier versions of this document. The list has now grown so long it needs its own subsection, which starts on page 19 of this document.

The Tsukurimashou fonts are distributed under the GNU General Public License, version 3, with an added paragraph clarifying that they may be embedded in documents. See the files COPYING and COPYING.GPL3, and note the following addition:

> As a special exception, if you create a document which uses this font, and embed this font or unaltered portions of this font into the document, this font does not by itself cause the resulting document to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the document might be covered by the GNU General Public License. If you modify this font, you may extend this exception to your version of the font, but you are not obligated to do so. If you do not wish to do so, delete this exception statement from your version.

The license means (and this paragraph is a general summary, not overriding the binding terms of the license) that you may use the fonts at no charge; you may modify them; you may distribute them with or without modifications; but if you distribute them in binary form, you must make the source code available. Furthermore (this is

where font-embedding becomes relevant) embedding the font, for instance in a PDF file, does not in itself trigger the source-distribution requirement.

My plan is that at some point in the future, when the fonts are in a more useful and complete form, I will make precompiled binaries available through commercial online channels. That will serve several purposes: it will allow me to make some money from my work, and it will also probably encourage some people to use the fonts who wouldn't otherwise. One of the bizarre aspects of human behaviour is that some people will buy a product they would not accept for free. Okay, whatever; in such a case I'm happy to take the money for it. Having a pay option will also give anybody who wants to support my efforts, an easy way to do that. For now, though, I am distributing Tsukurimashou only as this source package, with precompiled versions included for the Kaku and Mincho styles. The Jieubsida fonts, which don't need the in-progress kanji glyphs, may also be available in precompiled form as a separate package. If you want other styles, you'll have to compile them yourself or get them from someone who has done so. This limitation is deliberate: with the fonts in their current partial form, I'd rather limit their circulation to hobbyists.

Prior to releasing this package of source for the entire Tsukurimashou project, I released a smaller package of fonts covering just the Genjimon characters, in a different encoding. That package is now included in this one, but remains a separate entity. You will find it in the "genjimon" directory. It has its own, independent, documentation and build system. My plan is not to do much further maintenance on it; my hope is that not much further maintenance will be needed. If you want to use Genjimon characters, you have the choice of using the fonts from the Genjimon package (which encode the characters to replace the Latin alphabet) or using the full Tsukurimashou fonts, which encode the Genjimon characters in the Supplemental Private Use Area and also include a lot of other characters.

This documentation file gives some notes on the build system and on how to use the OpenType features built into the fonts. Other documentation files included in the package demonstrate what the fonts look like and list the current kanji coverage. Better documentation

(and some day, Japanese-language documentation) will probably appear in a later version; at the moment, I'm just more interested in designing fonts than in writing about them. Of course, all the typesetting in this manual is done with fonts from this package.

The name "Tsukurimashou" could be translated as "Let's make something!"

From time to time, people ask how they can help with the project. I'm hesitant to accept contributions to the coding, because of the pedagogical goal: I need to do it myself in order to learn by doing it. I also don't have much need for monetary donations. If you are in a position to actually offer me full-time employment appropriate to my skills and experience, I might like to hear from you, but as far as the Tsukurimashou project is concerned, the one thing that would really help a lot would be publicity. Share the link on social networks; write about it on your Web log (or invite me to write a guest posting); or even just do a "review" or a "rating" on the Sourceforge.JP project page. Tsukurimashou is also registered on Github, Ohloh, and CIA.vc; if you use one of those systems, you're encouraged to "follow" or "subscribe" to it as appropriate, both to keep yourself updated and to raise the project's profile.

There is some possibility that I may actually be able to get funding in the near future to work on kanji fonts and dictionaries full-time for a while. Exactly what that would mean for Tsukurimashou is unknown, because the project under discussion would have goals to but not exactly the same as the current Tsukurimashou/IDSgrep, and it's too soon to announce anything.

The home pages for this project, where you can download the latest releases, browse the source-control repository, and so on, are:

http://en.sourceforge.jp/projects/tsukurimashou/ (English)
http://tsukurimashou.sourceforge.jp/ (日本語)

よろしくおねがいします。

Matthew Skala
mskala@ansuz.sooke.bc.ca
June 18, 2012

0.6のニュース　What's new in 0.6?

The main visible content of the fonts hasn't changed much in version 0.6; I'm simply continuing the slog through the kyouiku kanji. This version contains the completion of Grade Three.

There's been a lot of less-visible work done, however, and that's one reason that this version has been so long in the making. This version contains its own bundled copy of the METAFONT-language portion of METATYPE1, and a Perl script that replaces the Awk portion of METATYPE1 and the wrapper provided by the mtype13 distribution. There's a story behind that.

I reinstalled the operating system on my laptop and so, in order to continue developing Tsukurimashou there, I had to also reinstall all Tsukurimashou's dependencies. I installed TEXLive, then tried to install mtype13 according to its instructions. Bear in mind that mtype13 is about ten years old and its installation process is designed for TEX distributions of that vintage, not present-day TEXLive. When invoked as directed, mtype13's Makefile automatically overwrote the master configuration that allows TEX and related programs to find their data. My entire TEX installation was rendered unusable. After reinstalling the entirety of TEX, I dug through the directory structure to figure out the actually correct place to put mtype13, and that fortunately wasn't hard. Thinking that I'd rather not lead Tsukurimashou users into making the same mistake I made, I figured I would add detailed instructions for installing mtype13 on a modern TEX, in this document.

But while researching that, I discovered that mtype13, although obviously intended to be free software, did not actually contain any license at all, making its status questionable. It also contained an outdated version of METATYPE1, and since I've been having trouble with METATYPE1 bugs right from the start of Tsukurimashou, it seemed like switching to a new version might be desirable.

I contacted the author of mtype13 and he very graciously released his decade-old code under an MIT-style free software license; but I wanted the updated METATYPE1, and in digging through that, I discovered that it was doing a lot of things not needed by Tsukurimashou, such as re-compiling all fonts twice, the second pass solely

to create the *.tfm files that I was throwing out anyway. Meanwhile mtype13 was doing things like writing output to /dev/tty, which was what drove me in an earlier version to put the whole works inside Expect so I could keep the build system output clean-looking. And METATYPE1 is public domain, and the parts used by Tsukurimashou amounted to just two files of METAFONT code plus an Awk script to process the output—the rest of the package is all about encodings, Latin Modern, back-translation from Postscript to METAFONT, and other things not used by Tsukurimashou. Then the only part of mtype13 really used by Tsukurimashou was a shell script that would run the Awk script.

All in all it seemed the logical thing to do was bundle an entire copy of METATYPE1's METAFONT code, and a script with function similar to mtype13 but with modifications appropriate to Tsukurimashou. That way I eliminate the dependencies on separately installed METATYPE1 and mtype13, and I have the opportunity to modify any features of those packages that aren't ideal for Tsukurimashou.

I combined the two METAFONT-language files into one and ripped out the second pass of font compilation; that alone doubles the speed of Postscript subfont generation. I also translated the Awk script into Perl, and removed all the hinting code from it. People who have the other dependencies of this package almost certainly have GNU Awk anyway, but I don't really want to try to maintain code in Awk, a language I barely know, and have that dependency for installation when I've already got a lot of Perl in the project. The automated Awk-to-Perl translator didn't work, for reasons unknown. Most of the Awk code seemed to be all about properly handling hints, which Tsukurimashou doesn't use at that stage of processing, so by removing all that stuff I ended up with a much shorter and cleaner Perl script than the original Awk.

Also worth mentioning is the "t1asm" utility, another previously unknown-to-me dependency of the mtype13/METATYPE1 complex. This is part of the popular "t1utils" package; I could have simply declared that package to be a new dependency, but there again, given that t1asm is just one small thing needed out of a larger package, and distributed under a permissive license, I decided to bundle it too.

The C code for this utility is in the t1asm/ subdirectory.

This version of Tsukurimashou is the first one designed to work with IDSgrep. IDSgrep is a separate, new project, with its own documentation and distribution packages, but its source code is checked into the Tsukurimashou SVN repository and so if you're following that respository, you should already have a copy of the IDSgrep code. IDSgrep brings the user-friendliness of grep to kanji search. It allows querying databases of kanji by partial descriptions of visual structure, such as "all kanji with 月 at the right." Tsukurimashou's build system now supports a "make eids" target, which will generate a database file suitable for querying with IDSgrep based on the visual structure of Tsukurimashou glyphs. Note that that may explicitly not be the same as their etymological origins; databases from other sources may be more useful for general dictionary purposes, but I've found that in Tsururimashou's own development a need exists for querying what is in the font as such, and IDSgrep exists in large part to serve that need.

外のプロジェクト　　Other similar projects

Maybe you shouldn't use this package! It is designed for specific purposes that are relevant to its designer, and although I certainly hope others will find it useful, my goals may or may not be in line with yours. Also, although I sometimes describe Tsukurimashou as the first parameterized METAFONT family with Japanese-language coverage, that claim requires careful qualification because many projects with similar aims have existed in some form for a long time. Here are some others, going back a few decades, that you might want to check out.

This is not intended to be a complete list; in particular, I'm leaving out many sources of CJK fonts that are not METAFONT-related, and many academic papers that are not associated with publically-available fonts. There is also no doubt a great deal of research and development locked up inside commercial organizations, or published in the Chinese language and thus inaccessible to me.

- "A Chinese Meta-Font," John Hobby and Gu Guoan, paper in

TUGboat 5–2, 1984. Proof of concept and discussion of some of the graphic design issues for parameterized CJK fonts. They built 140 radicals and 128 characters, using infrastructure very similar to Knuth's techniques for Latin fonts, and the traditional Chinese stroke-based analysis of characters. High-quality parameterized designs. No apparent plan to actually turn these into a usable full-coverage family; it seems to have been meant as research into the techniques only. http://www.tug.org/TUGboat/tb05-2/tb10hobby.pdf John Hobby has posted the source code described in the paper on his Web site at http://ect.bell-labs.com/who/hobby/hobbygu.tar.gz, but as he says in the enclosing Web page, the files are written in the now-obsolete METAFONT79 language and "they are of limited use because they are not compatible with today's METAFONT."

- pTEX, ASCII Corporation, 1987 onward: not a font project, but a TEX engine modified to handle 16-bit character codes and using existing fonts from other systems. Very popular in Japan; to some extent it still is, though other projects of similar nature (mostly not listed here) have gained a lot of market share in recent years.

- The Quixote Oriental Fonts Project, spearheaded by Dan Hosek, announced in a paper at the TUG 1989 Conference. Intended to be a parameterized METAFONT-native family for Chinese, Japanese, and Korean. Hosek apparently had some source code in hardcopy form displayed at the conference, but I've not been able to find the code nor any subsequent discussion of the project. http://www.tug.org/TUGboat/tb10-4/tb26hosek.pdf

- Poor Man's Chinese and Poor Man's Japanese, 1990, Tom Ridgeway: technology for displaying 24×24 bitmap fonts through METAFONT. This was not curve tracing, nor smooth scaling, but a way to actually display the dot matrix, jaggies and all. Still available in CTAN package "poorman," but considered obsolete.

- JemTEX, 1991, François Jalbert: included a program called jis2mf which would auto-trace 24×24 bitmaps to produce non-parameterized METAFONT code. Many sources from the 1990s (for instance, a

regular Usenet posting aiming to list all then-available .mf-format fonts) describe the availability of "Metafont for 61 Japanese fonts," which is the output from this program.

- The CJK package, described in TUGboat at least as early as 1997, still available though no longer popular, Werner Lemberg. Not a font project but a system for typesetting CJK text in LaTeX under the standard 8-bit engines, getting around the encoding issues by splitting each font into many smaller virtual fonts (similar to Tsukurimashou's "page" system). Fonts for this, at least at the outset, were usually bitmap fonts imported from other systems (one popular one was 48×48); later, as free vector fonts became available, those started to be used, some of them via auto-conversion from formats like TrueType to non-parameterized METAFONT.

- HanGlyph, 1997 and 2003: a language for describing Chinese characters, and support for rendering them in MetaPost and LaTeX. This is intended to address the Chinese equivalent of the "gaiji" problem: how to typeset rare characters that are not included in standard fonts or encodings. The user can describe the missing character and a small font containing just that character will be automatically created and used. In principle, HanGlyph's technology could be used to create a full-coverage font, but as of 2012 it doesn't appear anyone has done that. Availability and licensing terms are unclear; no released code or fonts seem to be available, but there have been papers published about it. http://www.hanglyph.com/

- IPA Mincho and IPA Gothic fonts, 2003 onward. Note "IPA" in this case stands for "Information-technology Promotion Agency," not the "International Phonetic Alphabet," and these fonts do not cover that IPA. Free high-quality fonts for Japanese, TrueType format, not parameterized. http://ossipedia.ipa.go.jp/ipafont/index.html

- XeTeX, SIL International, 2004 onward: TeX engine extended to handle Unicode and modern font technologies. Used to compile

this document, and one of the main compatibility targets for Tsukurimashou. http://scripts.sil.org/xetex

- Hóng Zì project by Javier R. Laguna. Aimed to be a parameterized METAFONT family for Chinese. The last release, which was in 2006, contained 125 characters. No infrastructure for addressing issues like METAFONT's 256-glyph limit, or radicals changing shape depending on their context. Probably abandoned. However, it did make several releases of code that you can still download and compile. http://hongzi.sourceforge.net/

- The KanjiVG Project, coordinated by Ulrich Apel, current in 2011. Still under development, but already has basically complete coverage, and is deployed in several important applications. This is not a font family, but a database of kanji (primarily from a Japanese point of view) broken down into strokes and radicals, with some curve points and a lot of abstract information about how the strokes correspond to the traditional radical classification (so that you can automatically recognize, for instance, what 示 and 礻 have in common), stroke order, and so on. This is a valuable resource for dictionaries and handwriting recognition systems. Some kind of supervised semi-automatic processing could probably turn it into a font, but keeping the style consistent (because the database has multiple sources), and adding the serifs and other visual information needed for styles like Mincho, would require some significant work. http://kanjivg.tagaini.net/ See it in action in Ben Bullock's handwritten kanji recognizer at http://kanji.sljfaq.org/draw.html.

- Character Description Language, from Wenlin Institute, Inc. Current in 2011. This is a commercial product. It is apparently (though the Web page could be clearer on the exact nature of what they're selling) a database of character descriptions similar to KanjiVG though with wider coverage, plus a binary-only rendering library, the combination available for license at some unspecified price. It says it's capable of generating MetaPost as one of its several output formats. Not clear to what extent there is parameterization, but presumably that would

be in the converter rather than in the resulting MetaPost. http://www.wenlin.com/cdl/

デベロップメントロードマップ　　Development roadmap

This version contains all the kyouiku kanji (the ones taught in Japanese elementary school) through Grade 3. The current plan is to release a minor version after each grade level of kyouiku kanji and one halfway through each grade level, which will take us up to version 0.12 at the end of Grade 6. Version 1.0 will probably be a separate version released shortly after 0.12, with a general clean-up and renovation, but it's possible I might skip directly from 0.11 to 1.0.

There are 1006 kyouiku kanji, though the fonts already contain more than that number of kanji glyphs because my general practice is to add other glyphs that are convenient to add whenever they come up, regardless of their level. For instance, when I added the "gate" radical 門 it was easy to add many other kanji that consist of that wrapped around an easy pre-existing kanji, even though some of those are not in common use and one, 閏, isn't even a real kanji at all, having been created by an error in the standards process. But having just over a thousand in the main-line roadmap makes the end of the kyouiku kanji a good milestone for the first major version number.

That may be a few years from now. Progress past that point will be somewhat dependent on how I feel about the project by then and what my personal career situation is. My hope is that at that point or before, I'll have the chance to present this work at one or more conferences and that it will have attracted some attention. Of course, if I can figure out a way to get paid to do it that would be nice, but attention is more important.

Although this is subject to change and cancellation, my current thought is that the next major versions would be 2.0 with the jouyou kanji (taught in high school, a total of 1130 additional glyphs), and 3.0 with the jinmeiyou kanji (the "name-only" kanji, 983 additional glyphs). I don't know how I'd break those up into minor versions, but presumably I'd aim for a similar spacing of about 100 new characters per release. At the 3.0 point, with a little over 3000 kanji, the fonts

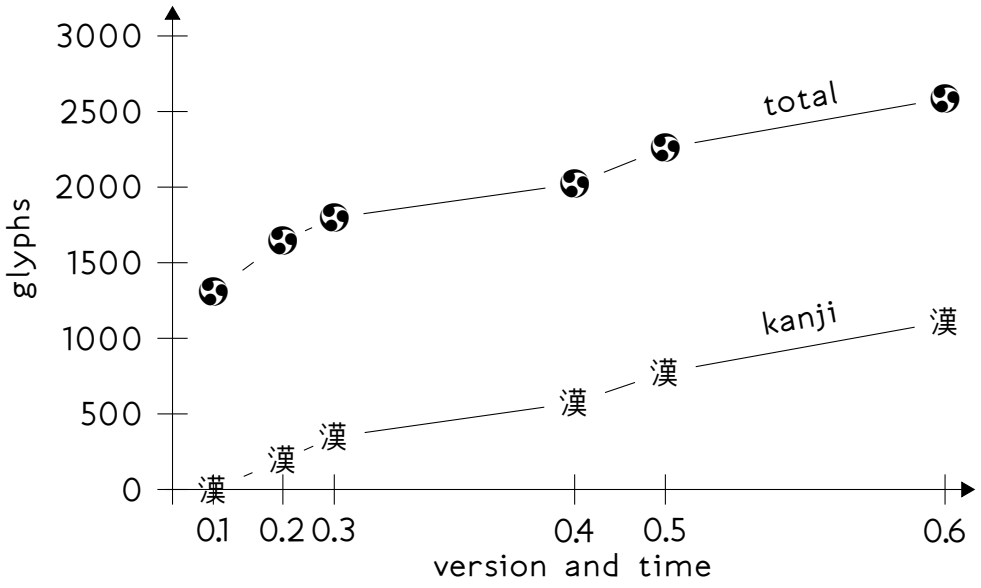


Figure 1: Growth of glyph counts

should be basically complete in the sense of being usable to write the full Japanese language as most reasonably competent native readers know it. Many more kanji exist; I don't know how far I'll want to take this project toward covering them all. For reference, looking at some other fonts I have handy, IPA Mincho contains 6682 kanji (probably aiming to cover the JIS 208 standard), and Sazanami Gothic contains 12202 (probably aiming to cover JIS 212). Those might be reasonable milestones for 4.0 and 5.0.

Figure 1 is a chart of the progress to date. Note the horizontal axis is labelled by version but scaled by time. Even-numbered versions tend to take more time because I tend to do the easier characters in each grade level first. All the glyph counts in these charts are for the Tsukurimashou (Japanese) fonts alone. The Jieubsida (Korean) fonts contain the 11172-glyph block of precomposed syllables, which because they are algorithmically generated cannot be well compared to the more manually-created kanji and other glyphs. The Jieubsida fonts also contain a few hundred non-precomposed glyphs, beyond the core they share with Tsukurimashou.

Figure 2 gives a different view of development progress: the number of lines of code (total lines in mp/*.mp, including comments and blanks

but not including code in other languages and locations; also excluding jieub-*.mp, but still including a few other files from Jieubsida) plotted against the total number of glyphs in the main Tsukurimashou family. In version 0.6, there are a few thousand lines of code added due to the new bundling of METATYPE1, which may skew the numbers a bit.

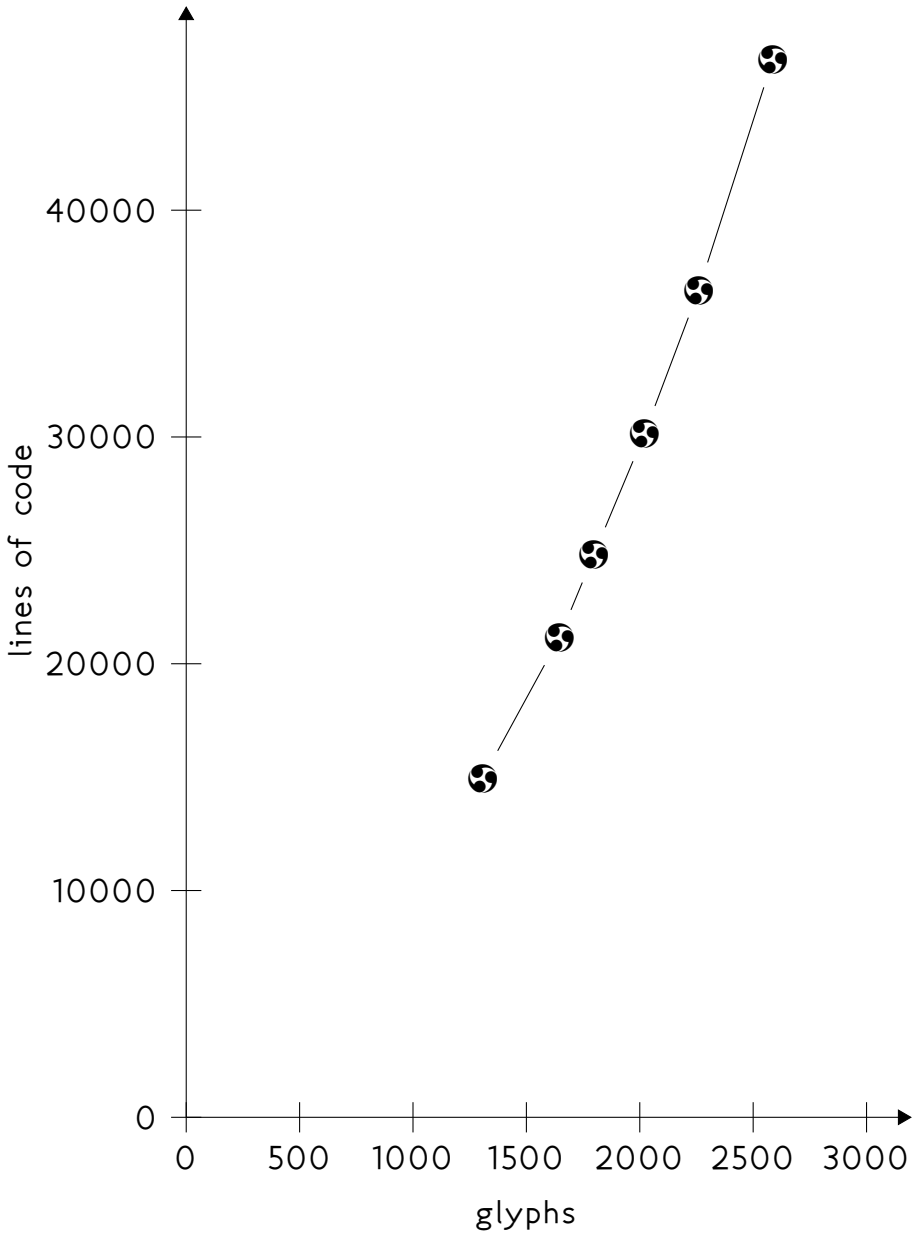


Figure 2: Lines of code per glyph

外のソフトのバグ　　Relevant bugs in other software

Building Tsukurimashou (and, especially, its documentation) requires the use of some fairly advanced features of third-party software. Some of those features are not often used; as a result, I've become an unintentional beta-tester for the third-party packages. I've previously noted bugs when they come up, in code comments and the relevant parts of this document, but as of the current version, such bugs have become so numerous that it makes sense to also have a central list.

- Metapost (maybe even METAFONT) issue propagated to METATYPE1: the equation solver has a non-renewable resource of "independent variable instance serial numbers" which are consumed as code executes. Basically, one is used up permanently every time an assignment statement executes. Very old versions of Metapost either did not have these, or allowed the counter to wrap around, and so the solver would produce incorrect results in long-running programs. At some time before version 0.641, the solver mistakes were fixed by the limited serial-number scheme, but serial numbers would run out and cause a fatal error when they reached 2^{25}. In version 1.501, the limit was increased to 2^{31}. Any sufficiently long-running Metapost program will eventually die as the limit is exceeded. Some experimental versions of the Blackletter Lolita curve-fitter would exceed the 2^{25} limit; the current experimental version is less computation-intensive, probably wouldn't exceed 2^{25}, and certainly fits comfortably in 2^{31}, but it seems like the limit should not exist at all. Debugging is hindered by some currently in progress redesign work on Metapost's data structures, such that (as of October 2011) the cutting-edge development version leaks memory and crashes for that reason long before the serial numbers can run out anyway. Memory leaks acknowledged and planned to be fixed by Metapost maintainer Taco Hoekwater, but it may take a while. I've posted a link target to track this issue at http://ansuz.sooke.bc.ca/entry/213.

- METATYPE1 sometimes runs glyph names through the METAFONT tokenizer. At this point, I don't know how essential

that is to the operation of METATYPE1 or whether it can be changed. It has subtle effects that can cause problems. One issue shows up in glyph names that contain a decimal digit followed by a dot, as in "uni1100.bug"; then what gets written into the Postscript output is "uni1100bug" because that is equivalent but more canonical in METAFONT syntax. A more serious issue shows up with the glyph name "uni1100.l1" from Tsukurimashou 0.5; in the new METATYPE1 version 0.55, that gets tokenized in a context where the token "l" is a "spark," and so the whole compilation fails. The workaround is to change "l" to "lj" (for "lead jamo"). We are not actually using the glyph names written into the Postscript files, so it doesn't matter if they are incorrect, but compilation must not fail. In the longer term it would be nice to make METATYPE1 not exhibit this behaviour, and pass the strings intact into the Postscript file.

- METATYPE1 pen_stroke_edge macro: as of METATYPE1 version 0.44, if left to its own devices it will sometimes attempt to evaluate the "turning angle" of a zero vector, and then blow up. This seems to happen most often when stroking a vector in a direction of approximately 290 degrees. As of Tsukurimashou 0.6, we are bundling METATYPE1 version 0.55, which seems to have fixed this bug; the workaround in previous versions of Tsukurimashou has been removed.

- METATYPE1 infrastructure in general: sometimes generates paths that METAFONT calls "degenerate," triggering a fatal error. Workaround is to filter things, before rendering, through the regenerate macro in intro.mp, which removes any very short path segments. Possibly related: the Fill macro will sometimes abort in response to some conditions on "turning number" that do not appear to actually be harmful. Workaround is to use our dangerousFill in intro.mp instead, which is just a copy of Fill with the error checking removed.

- FontForge spline geometry operations, such as overlap removal and simplification: these have historically tended to be very numerically unstable, and subject to some combination of infinite loops, segmentation faults, unexplained floating-point exceptions,

bizarre error messages, incorrect output and so on. The most recent development versions seem to be relatively good; best advice is to switch to one of those, turn debugging on, and hope. It is also sometimes possible to get past individual problems by switching between the different floating-point formats offered by FontForge's compile-time configuration: float, double, and long double. But which of those is the best choice isn't always predictable. Some numerical tweaks in actual glyph outlines also exist to try to work around these issues.

- FontForge usually fails to automatically insert appropriate sub-table breaks when it reads a feature file describing a large OpenType table. Workaround: our Perl code inserts explicit breaks; trial and error needed to figure out how frequent they ought to be.

- When FontForge saves an Adobe feature file, what it writes may bear only a passing resemblance to what was actually in the font, and in particular, cannot subsequently be loaded by FontForge (sometimes cannot be loaded at all, other times can be loaded but the loaded tables behave differently from the ones that were saved). Workaround: do not save feature files from FontForge.

- FontForge can segfault while writing a feature file because of dereferencing a pointer first and checking whether it was null afterward. Fixed by developers in December 2011.

- FontForge does not apply features of the "DFLT" language system as a default to Unicode ranges that have no explicit mention in the font file (Adobe spec says it should). Workaround: explicitly list language systems to cover every Unicode range we care about.

- FontForge may write up to 23 horizontal stem hints per glyph when writing a PostScript-flavoured OTF font, resulting in a PostScript stack overflow on loading if the glyph also has a non-default advance width. Went undetected for a long time apparently because only CJK fonts are likely to have so many

horizontal stems, and CJK fonts are likely to be monospace and thus won't also have per-glyph non-default advance widths. Fix: change the limit to 22. Patch offered on FontForge mailing list in November 2011 (http://sourceforge.net/mailarchive/forum.php?thread_name=alpine.LNX.2.00.1111231635070.28210%40tetsu.ansuz.sooke.bc.ca&forum_name=fontforge-devel), applied by FontForge developers in December 2011.

- FontForge save and then load of an SFD file has the effect of renaming the "clig" feature to "rtla" on only one of the three machines where I've tried it. May be related to the x86-64 architecture. Reported on mailing list November 2011, mentioned in a developer's commit message December 2011, whether it is in fact fixed is unknown.

- FontForge rasterization to BDF via FreeType as opposed to whatever other code FontForge would use: sometimes produces corrupt results. Reported to FontForge mailing list in November 2011, inspiring FreeType maintainer Werner Lemberg to find and fix an unrelated bug in FreeType (it was unable to process BDF files with high code points; see http://savannah.nongnu.org/bugs/?34896). But that bug did not actually affect us because FontForge uses its own code for BDFs instead of FreeType's anyway, and Lemberg says can't help with the corrupt rasterization. Workaround is to compile FontForge without FreeType support; from GUI it is possible to just turn off FreeType rasterization on individual bitmap-creation operations, but that option doesn't seem to be available from the scripting language and so the package has to actually be built without FreeType. As of April 2012, seems not to be an issue with Arch Linux's packaged versions.

- XeTeX fails to advance glyph pointer after a successful match in a GSUB table, which has complicated consequences for chaining substitutions, most notably that "ignore sub" rules have no effect. This is actually a bug in the third-party ICU library which XeTeX uses. Reported to XeTeX mailing list in November 2011; after discussion there (http://tug.org/pipermail/xetex/2011-November/022298.html) found existing issue in ICU's bug

tracker dating from June 2010 (http://bugs.icu-project.org/trac/ticket/7753). Poor workaround is to carefully write all substitution features to work regardless of whether the pointer advances; one useful technique is to make what would be an "ignore sub" rule instead substitute to a series of identical-looking glyphs that are never matched on input to subsequent rules.

- XeTeX fontspec package: in some versions has trouble with treating its WordSpace configuration option as a multiplier when font size changes, in a way that is most noticeable when using monospaced OTF fonts; complicated by the fact that it's hard to define just what should count as a "monospace" font. Discussed at length on the XeTeX mailing list in February 2011 (http://tug.org/pipermail/xetex/2011-February/020065.html) and eventually resulted in three items and planned fixes in the fontspec issue tracker (https://github.com/wspr/fontspec/issues/97, https://github.com/wspr/fontspec/issues/98, https://github.com/wspr/fontspec/issues/99). The main bug has now been fixed, and it has also become less relevant to Tsukurimashou documentation since the introduction of the proportionally spaced fonts, but still bears some watching. If they implement all my suggestions, then the current PunctuationSpace multipliers in Tsukurimashou documentation will become much too large and need to be reduced. Earlier versions of Tsukurimashou toyed with workarounds based on the everysel package or on poking into the internals.

- XeTeX and fontspec do not apply by default some OpenType features that are supposed to be applied by default (in particular, "ljmo" and "vjmo"). Workaround: manually request them with the RawFeature option.

- LaTeX tocloft package: as of May 2011, it sets an entry without a dot leader by actually requesting an entry with a leader, but with a font-dependent invalid spacing between dots. The DVI renderer is supposed to reject the bad spacing and not set a leader at all. With default font sizes, in most DVI renderers, that results in either the correct appearance (no leader) or an almost-correct appearance (a single dot instead of a leader). But

with the larger sizes of the extsizes classes, in some renderers, the resulting DVI file is so invalid as to cause the renderer to blank the rest of the page. Reported to Will Robertson, package maintainer; he acknowledges it, and says will fix (by not requesting dot leader when they're not wanted) in next version. A workaround involving poking into the package internals to set a smaller invalid spacing is implemented near the top of doc/bkstyle.tex.

- PGF (lower layer of TikZ) shapes.callouts library: in version 2.10 only, ellipse callouts just don't work (causing fatal TEX errors if attempted), apparently because a macro name was changed in PGF development process and the change not propagated to all files. Workaround implemented in our build system consists of detecting version 2.10 and making a local modified copy of the buggy file with the macro name corrected. Problem is apparently known to the maintainers and already fixed in their source code repository before I noticed it, presumably will not be an issue in any future releases, but buggy 2.10 is latest release and widely used as of late 2011; it will probably remain in the wild for a long time.

『作りましょう』の使い方
Using Tsukurimashou

Fonts appear in OpenType format in the otf/ subdirectory of the package. If you just unpack a distribution, there will be four fonts there corresponding to the monospace and proportional versions of Tsukurimashou Kaku and Mincho. If you run the build process, that will create more. For a quick start, all you need to do is install the font files in whatever way is standard for installing OpenType fonts on your system. It is safe to delete everything else in the package, though you might want to keep the PDF documentation files.

Samples of what the different styles look like are in the file demo.pdf, which see. Here's a brief summary:

- 作りましょう角 Tsukurimashou Kaku ("Square Gothic"): sans-serif with squared stroke-ends.
- 作りましょう丸 Tsukurimashou Maru ("Round Gothic"): sans-serif with rounded stroke-ends.
- **作りましょうアンビル的 Tsukurimashou Anbiruteki ("Anvilicious"): extra-bold, rounded sans-serif.**
- 作りましょう天使の髪 Tsukurimashou Tenshi no Kami ("Angel Hair"): very thin hairline display font.
- 作りましょう僕っ娘 Tsukurimashou Bokukko ("Tomboy"): felt marker style.
- 作りましょう明朝 Tsukurimashou Mincho ("Ming Dynasty"): modern serif.
- *ツイタ頭 TsuIta Atama ("Head"): italics to go with Tsukurimashou Kaku.*
- *ツイタ足 TsuIta Soku ("Foot"): italics to go with Tsukurimashou Mincho.*

The remainder of this document is reference material describing the special features of the fonts, as well as instructions on how to build and customize your own font files, including the styles that aren't distributed in precompiled form.

OpenTypeのフィーチャー
OpenType Features

OpenType contains a mechanism for defining what are called "features" in a specific technical sense rather than the more general usage of that word. They are identified by four-letter tags, and generally correspond to extra data added to the font that compatible renderers can interpret to provide special typesetting effects. The Tsukurimashou build system's configure script accepts an "--enable-ot-features=" argument which can be given a list of feature tags, or "all", each optionally prefaced by an exclamation point to negate it. These are processed left to right, so that a setting like "--enable-ot-features=all,!ccmp" will enable everything except the "ccmp" feature. The default value is "all."

Be aware that just because you selected a feature during config does not mean you can actually use it. Features are only included in fonts if they make sense for the particular font being built (thus, monospace fonts will not contain proportional-only features) and if all necessary character codes are available (thus, fonts with Japanese glyph coverage will not contain shaping features specific to Korean script). Also, build-time configuration only controls what will be included in the font. Given that the feature is in the font, your renderer must then support the feature you want to use, and you may need to adjust the renderer configuration to tell it to use the feature. Some renderers make that easier or harder to do than others, and some renderers do not turn on by default certain features that I recommend and the standards say should be turned on by default.

Note that although OpenType would permit such a thing, features in any single Tsukurimashou font are NOT language- or script-specific; for instance, the set of features available for Latin script is the same as the set of features available for Japanese script in the same font, and the behaviour of each feature does not change depending on whether the text is tagged as "English" or "Japanese." This even extends as far as, for instance, the fact that you can really kern kanji in Tsukurimashou proportional fonts. A feature applied to a code point sequence for which it has no meaning (such as "lead jamo

shaping" applied to English text) will simply have no effect. Specific languages are nonetheless mentioned in the font tables for the benefit of FontForge, which doesn't seem to handle defaults very well, and any other software that behaves like FontForge.

外の分すう　　Alternate Fractions (afrc)

Tsukurimashou does not contain any special support for diagonal fractions, but it does support vertical or "nut" fractions using the OpenType feature "afrc." With the afrc feature turned on, any sequence of up to four digits, a slash, and up to four more digits becomes a vertical fraction:

1/2 → ½　　34/56 → 34/56　　789/123 → 789/123　　4567/8901 → 4567/8901

This feature works with both the "narrow" digits and slash (ASCII code 47–57, Unicode U+002F–U+0039) and the "wide" ones (Unicode U+FF0F–U+FF19). If the input digits are narrow, and the fraction is one digit over one digit, then the resulting glyph will be narrow (the same width as monospaced Latin characters). Otherwise—with wide input or more than one digit in the numerator or the denominator—the fraction will be the width of a wide (ideographic) character.

]1/2[→]½[　]１／２[→]½[　]34/56[→]34/56[　]３４／５６[→]34/56[

As of version 0.5, this feature works in both the monospace and proportional fonts. However, in order to make it work in proportional fonts it was necessary to exclude the glyphs involved from the kerning table; as a result, only precomposed fractions will benefit from kerning.

キャプがスモールになって　　Capitals to Small Caps (c2sc)

This feature replaces capital letters with small caps, as might be desired for typesetting all-caps abbreviations like TEPCO and IAEA. It works much like the "smcp" feature. Similar limitations apply.

グリフの併合と分解　　Glyph (De)Composition (ccmp)

This feature, which only exists in the Jieubsida fonts, breaks precomposed syllables that don't have final consonants into their component jamo, and combines sequences of jamo for single vowels and consonants into jamo representing the sequences, where possible. Splitting tail-less precomposed syllables is necessary to support the Unicode behaviour of adding a tail to a precomposed syllable that doesn't have one; the splitting is later undone by the "liga" feature. Combining single jamo into multiples does not seem to be required by Unicode, but is vaguely described in Microsoft's spec, and it seems like a reasonable thing to do. It is recommended that this feature should be turned on by default in fonts that have it, and that is required by Microsoft's spec.

合字　　Ligatures (liga)

The Tsukurimashou fonts do not contain ligatures for the Latin script at present. Their basic letter forms were designed with monospace setting in mind, where ligatures don't make sense; and given the importance of keeping everything legible to readers whose native language is Japanese and who may not be familiar with some of the advanced aspects of the Latin script, it was a design decision to make the letters look good without ligatures even in proportional fonts.

The Jieubsida fonts, however, contain an extensive ligature table for hangul, and use of this feature is required to get full support for hangul script. Without it, the decomposition of precomposed tail-less syllables in the "ccmp" feature will stand, leaving those syllables looking poorly designed; and syllables written out as individual jamo will be approximated with on-the-fly composition even when a precomposed glyph would be available. In order for the ligature table to operate correctly, the "ljmo" and "vjmo" features should be turned on. See the section on Korean language support for more details of how this feature works.

頭子音の形　　Lead Jamo Shaping (ljmo)

This feature only exists in the Jieubsida fonts and should be turned on by default. It replaces "lead" jamo glyphs (consonants at the starts of syllables) with contextual variants that depend on the vowel and

the presence or absence of a tail. For more information, see the section on Korean support.

文字に付け方 Mark to Base Positioning (mark)

This feature only exists in the proportionally spaced fonts; a similar effect is achieved in monospace fonts as a consequence of the way the glyph side-bearings work. It should be turned on by default. The feature stores data for OpenType renderers to attach accents to letters, allowing for combinations of letter and accent that do not have precomposed glyphs of their own. For instance, U+0071 U+0306 will generate "q̆," a lowercase q with a breve, which does not correspond to any Unicode code point and would not otherwise be available.

As of version 0.6, support for this feature is limited. Not every base letter you might want will necessarily be available; all the most popular combining diacritical marks exist, but it's easy to imagine others you might want that are not included; in many cases (especially when capital letters are involved) the mark-to-base version of a combined character ends up looking significantly different from the precomposed version; some of the combinations just don't look very good; and intended extensions of the scheme to, for instance, allow adding dakuten to Japanese kana don't exist yet at all. But many common cases should be covered.

点に付け方 Mark to Mark Positioning (mkmk)

This feature works in combination with the "mark" feature above, and it only makes sense to use when "mark" is turned on; like "mark" it is only available in proportional fonts. It stores data used by OpenType renderers to add more marks to glyphs that already have some marks; in particular, this should allow stacking up more than one accent on the same letter to create many glyphs that would be impossible by other means.

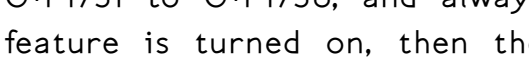
In version 0.6, this feature has been expanded to cover a few more of the Latin accents than before, allowing the construction of such characters as x̌. It is still somewhat experimental, however.

メタデータ　　Metadata Table (name)

This is not the usual kind of OpenType feature, but it can be turned on and off through the same build-system interface. By default, the font will contain a "name" table listing metadata such as creatorship and licensing, in both English and, as appropriate, one of either Japanese or Korean. Some of this information is obligatory, so if you turn off the "name" feature you actually still get a name table, but it will be less completely populated.

巴の花形　　Ornaments (ornm)

Tsukurimashou provides eight tomoe ornaments that look like this:

These glyphs are encoded to the Unicode private-use code points U+F1731 to U+F1738, and always available that way. If the "ornm" feature is turned on, then they will also appear as substitutions (not alternates!) for the ASCII capital letters A through H; so that the text "A BIG TEST" comes out as "  ." It is likely that the way this OpenType feature works will change in the future, since it seems not to be current best practice to implement ornm by substitution but I'm not quite sure yet what the best practice actually is.

These ornaments look the same in all the fonts; they do not change from one style to the next.

FontForgeだけのメタデータ　　FontForge-specific Metadata (pfed)

This is not the usual kind of OpenType feature, but it can be turned on and off through the same build-system interface. When selected, it causes the build system to add a "pfed" table with a "flog" subtable in the generated OpenType fonts, containing verbose information about the installation on which the fonts were built. This table is a FontForge-specific extension of OpenType format; the name "pfed" refers to PfaEdit, an earlier name of the software that later became FontForge. Other software will ignore this table; but users with FontForge can examine the verbose metadata under the "FONTLOG"

heading in the "Font Info..." dialog. The data is also visible as a chunk of plain text near the end of the file when the OTF file is examined with a general-purpose file viewer such as "less." The main interesting content is the command line that was given to configure, and a dump of most of the variables known to Make. There's also a copy of the copyright notice and URLs for the project home pages, giving in more verbose detail some of the same information included in the "name" table.

I recommend activating this feature, especially if you will be building fonts for distribution to others. It may make bug reporting easier, because it means that anyone who gets a copy of the font also gets some information on where that font came from; if someone builds a font and then has trouble with it, it'll be easier to help if the font contains this debugging information. It also improves the chances that should a font file get separated from its context, someone stumbling upon it will be able to figure out what to do with it. The Net is full of inaccurately labelled fonts with unknown authorship, often being sold by shady commercial enterprises that have no legal right to do so, and we all ought to do what we can to stamp that out. Detailed human-readable metadata, in general, is a Good Thing.

But the font log contains information like software version numbers, user account names, and installation directory names. Some people have funny ideas about the sensitivity of such information in relation to system security; they may think that revealing it creates a real risk, not otherwise present, of people breaking into their computers. They might also think it could be used to trace the origins of anonymously written PDF files and that forensic investigators don't have many other ways to do that.

Such people are wrong. However, my saying so won't cause them to change their minds. If I distributed software that attached this information to generated fonts by default, then someone who didn't read the documentation would eventually "discover" it and make a big fuss about it supposedly being a security hole. Who needs that? The feature is therefore turned off by default. I recommend turning it on by passing the "–enable-ot-features=all" option to configure. The

default is "–enable-ot-features=all,!pfed," which enables everything else.

Regardless of the setting chosen, the build system will place the same information in a file called "fontlog.txt" in the txt/ subdirectory of the build tree. After doing a build you can read that file to see what would have been put in the fonts. The option setting just controls whether or not the fontlog file will be added to the OTFs during the final packaging step.

スモールキャピタル　　Small Caps (smcp)

Small caps are available through the OpenType "smcp" feature. This feature simply substitutes the 26 lowercase ASCII Latin letters with alternates that are encoded into the Private Use Area at the code points formerly used by Adobe for this purpose: U+F761 to U+F77A. At some point in the future, other glyphs may be added to support small cap letters other than the 26 ASCII ones, and I may start using unencoded glyphs rather than mapping them into the PUA.

ヘビーメタルウムラウト　　Heavy Metal Umlaut (ss01)

With Stylistic Set 1 (OpenType feature "ss01") turned on, umlaut or dieresis over the vowels ÄËÏÖÜŸäëïöüÿ as well as tilde over Ñ and ñ is replaced by a "heavy metal" umlaut intended for spelling musical names like "Motörhead," "Spın̈al Tap," and "Mormon̈ Tabärnacle Choïr." The heavy metal umlaut differs from the regular umlaut in that the dots are larger and pointier.

Glyphs to support this feature, including a "spacing heavy metal umlaut" character, are encoded into the private-use area at U+F1740 through U+F174E.

丸つき字　　Enclosed Letters and Numerals (ss02)

With Stylistic Set 2 (OpenType feature "ss02") turned on, the enclosed characters described by Unicode become available as contextual substitutions for sequences of (in most cases ASCII) characters:

- (0) → ⓪ through (50) → �l⓪

- (A) → Ⓐ through (Z) → Ⓩ
- (a) → ⓐ through (z) → ⓩ
- (ア) → ㋐ through (ン) → ㋓
- ((1)) → ⑴ through ((10)) → ⑽
- {0} → ⓿ through {20} → ⓴
- {A} → 🅐 through {Z} → 🅩
- [A] → 🄰 through [Z] → 🅉
- <A> → 🅰 through <Z> → 🆉

The choice of which ranges of numbers and letters to support is mostly not mine—I just implemented what I found in the Unicode charts.

Unicode only has code points for the unvoiced versions of the enclosed katakana (e.g. ㋕㋖㋗ but not enclosed versions of ガギグ), and it has no code point for enclosed ン. I've added ㋓, encoded to private-use code point U+F1711, but not the others. If there's a demand for other enclosed characters, though, they are pretty easy to add. I held off on just defining dozens or hundreds more, because it's not clear to me what the typical use of these characters actually is, and a complete set of enclosed characters might be better seen as a font in itself rather than a series of special characters inside the font.

Note that for the enclosed katakana the substitution will accept either ASCII parentheses or wide parentheses (U+FF08 and U+FF09); the others work with ASCII parentheses only.

母音の形　　Vowel Jamo Shaping (vjmo)

This feature only exists in the Jieubsida fonts, and should be turned on by default when it exists. It replaces "vowel" jamo glyphs with appropriate contextual variations depending on the layout of the syllable. The layout that will be used by "vjmo" is actually chosen during execution of the "ljmo" feature, which must be applied before

"vjmo" (as required by Microsoft's specification) for "vjmo" to work properly. Note that Microsoft also describes, but we don't use, a similar "tjmo" feature for reshaping the tail.

付加文字
Extra glyph coverage

げんじもん　　Genjimon

Glyphs for the 54 Genjimon are encoded to private-use code points U+F17C1 to U+F17F6, in the order of the corresponding Tale of Genji chapters (“Kiritsubo” to “Yume no Ukihashi”). The style of these glyphs varies a fair bit between the different font styles. Here are samples:

- Kaku: 
- Maru: 
- **Anbiruteki:** 
- Tenshi no Kami: 
- Bokukko: 
- Mincho: 

A set of Genjimon-only fonts, derived from the Tsukurimashou code, exists as a spin-off project in the genjimon subdirectory of the Tsukurimashou distribution. The documents there contain some more information about the history of these symbols and what they signify.

えききょう　　I Ching

All the I Ching-related characters defined by Unicode in the Basic Multilingual Plane are supported: trigrams U+2630 to U+2637, monograms U+268A and U+268B, digrams U+268C to U+268F, and hexagrams U+4DC0 to U+4DFF. These characters are sized to fit in the same box as kanji, and so in most cases you will probably want to scale them to a significantly larger point size than that of nearby text. The style does change somewhat from one font to the next.

There are also some “I Ching dot” combining characters in the private-use code points U+F1701 to U+F1709, intended to show movement of

lines 1 through 6 of hexagrams or 1 through 3 of trigrams, encoded in that order. The idea is that you can typeset something like this:

䷀:→䷁

using a sequence of codes like this:

U+4DDE U+F1702 U+F1704 U+2192 U+4DEF

解字の文字　　Ideographic Description Characters

Unicode defines a set of special code points (U+2FF0 to U+2FFB) and a syntax for using them to describe what Han-script characters look like. For instance, 「歯」 might be described by the string 「⿱止⿴凵米」, which expresses that 歯 consists of 止 written above a composite sub-glyph that itself consists of 凵 wrapped around 米. Unicode's scheme for character description appears to have been inherited more or less unmodified from a similar scheme in the Chinese GBK standard.

Tsukurimashou includes glyphs for the special characters. There is also support in the build system for a "make eids" target, which generates a file called tsukurimashou.eids in the txt/ directory. That file is a collection of Extended Ideographic Description Sequences (EIDSes) describing the construction of the kanji in the font. This is the interface to IDSgrep, a program for searching kanji according to powerful criteria on their visual structure. IDSgrep is available from the same SourceForge.JP project that hosts Tsukurimashou; see that package and its documentation for more information on EIDS format and how to use it.

With or without IDSgrep, Tsukurimashou may be used to typeset the special characters for the descriptions. The glyphs look like this: ⿰⿱⿲⿳⿴⿵⿶⿷⿸⿹⿺⿻

公のユーログリフ　　Official Euro Sign

Fonts that include a symbol for the European currency unit (Unicode U+20AC) usually re-draw it to match the style of the rest of the font, and the Tsukurimashou fonts are no exceptions. However, there is an official glyph design that apparently was intended to be normative—in theory, you're supposed to use the official glyph, which does not

change with the style of the rest of the font, even if it clashes. For those who want to do so, the official Euro sign is included in the Tsukurimashou fonts at private-use code point U+F1710.

It looks like this: €

ハングル語
Korean Language Support

First, please note that in general (with some exceptions) I follow the practices of Unicode for such things as the names of letters, the name of the writing system itself, and so on. Naming decisions are thought by some people to have political implications. It is not my purpose here to take a position on any issues except technical ones of font design; but for practical reasons I must settle on some set of names.

The Tsukurimashou project is generally focused on the Japanese language; but Korean has some degree of connection to Japanese, the hangul writing system used for it is technically interesting and superficially appears easy, and for various reasons, I decided to extend Tsukurimashou to include some support for Korean. The result is the "Jieubsida" 「지읍시다」 series of fonts. The name is as nearly as possible a direct translation from Japanese to Korean of the name "Tsukurimashou" 「作りましょう」. Jieubsida builds from the same code base as Tsukurimashou, if you select the appropriate options during configuration.

There are several ways to write Korean in Unicode, and they are supported to varying degress by Jieubsida:

- Individual jamo from the "hangul compatibility jamo" range of code points, U+3131 to U+318E.
- Precomposed syllables from the "hangul syllables" range, U+AC00 to U+D7A3.
- Conjoining jamo from the "hangul jamo" range, U+1100 to U+11FF, and its supplements.
- Hanja (Han Chinese characters) from the "CJK unified ideographs" range, U+4E00 to U+9FFF, and its supplements.

Hangul is theoretically a purely phonetic alphabet. Letters are called jamo, and each one is supposed to correspond to exactly one phoneme

of the Korean language. The set of jamo in Unicode and Jieubsida is a little bigger than in the standard present-day Korean language, as a result of historical changes in the centuries since the writing system was introduced. The rules for how jamo may be used together have also become more restrictive.

Words are written divided into syllables, with each syllable arranged according to certain rules to fit into a square box—much like the square box used per character in writing Japanese or Chinese, but with Korean, there are multiple jamo in each box, instead of one kana or kanji per box, and the question of whether to count each jamo as a character or each box as a character is the start of the fun.

Every syllable consists of a "lead" containing between one and three jamo, a "vowel" containing between one and three jamo, and a "tail" containing between zero and three jamo. There are "consonant" and "vowel" jamo; the lead and tail consist exclusively of consonant jamo and the vowel consists exclusively of vowel jamo. Leads, vowels, and tails are not allowed to be just any combinations of the right number of the right kind of jamo; there are relatively short lists of possible leads, vowels, and tails.

In the relatively standardized present-day form of the Korean language, there are 19 different leads, 21 different vowels, and 28 different tails (including the empty tail of zero jamo), and only two of these (the vowels 「ㅛㅐ」 and 「ㅝㅔ」) contain more than two jamo. One of the 19 leads actually corresponds to an empty or silent lead with no sounds in it, but the empty lead is written using the single jamo 「ㅇ」, which is not otherwise allowed as the lead; thus the lead is always written with at least one jamo, and in the relatively standardized present-day language, at most two. Also, two-jamo leads on the list of 19 always consist of one jamo repeated twice (not two different ones); some two-jamo tails contain two different jamo. Combinations of more than two jamo, and other single and double jamo not on those lists, occur in archaic contexts, less-popular dialects, and so on; but there are very many of those longer, or merely other, combinations defined by Unicode.

コンパチのジャモ　　Compatibility jamo

The “hangul compatibility jamo” range is fully supported by Jieubsida. It’s easy: one jamo per character box. But it is not done to actually write the Korean language that way; readers and writers want one syllable per character box, with the individual jamo changing size and layout to pack nicely. As a result, the “compatibility jamo” glyphs are basically only useful in documents like this one, which discuss the technical details of the writing system.

Hangul compatibility jamo look like 「ㅈㅣㅇㅡㅂㅅㅣㄷㅏ」. Writing words that way leads to some ambiguity in syllable division, which may be one reason it never caught on (though the same ambiguity exists in most Romanization forms).

併合のシラブル　　Precomposed syllables

The Unicode standard takes the 19 leads, 21 vowels, and 28 tails of the relatively standardized present-day language and multiplies them together to get 11172 possible syllables. Each of those syllables has its own code point in the “hangul syllables” range. A font with a glyph for each one, or even just a glyph for each of the fraction of them that actually occurs in present-day relatively standardized Korean, can be used to write the present-day language with a minimum of fuss. That’s what most people do; most Korean documents on the Web are encoded into that range of Unicode; and Jieubsida contains the full set of glyphs to support it.

The precomposed syllables look like 「지읍시다」.

併合のジャモ　　Conjoining jamo

If you want to represent Korean syllables that contain leads, vowels, or tails other than those in the Unicode precomposed syllables, or if you want to be able to process text at a sub-syllable level (which might be useful for input methods, among other things), then you may end up dealing with the “conjoining jamo.” Unicode attempts to have a code point for every possible lead; one for every possible vowel; and one for every possible tail, including separate lead and tail code points in the fairly common case of the same combination of jamo being allowed as both a lead and a tail. You are supposed to be able

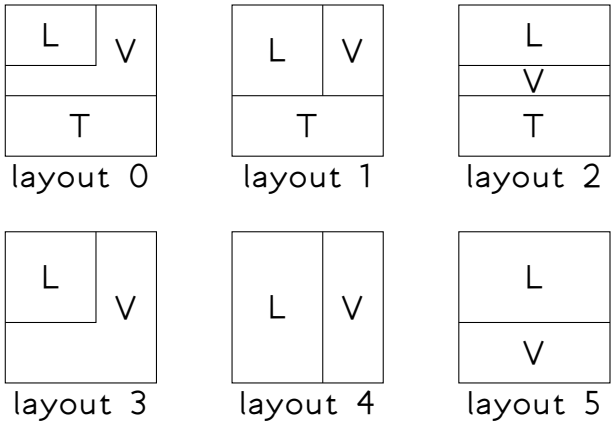
to write your documents using those code points, and then magic beyond the scope of Unicode is supposed to typeset them properly. Unicode defines a "canonical equivalence" between the precomposed syllables and the conjoining jamo, where every precomposed syllable can be interchanged with the conjoining jamo from which it's made. It is even supposed to be possible to take a precomposed syllable with no tail, follow it by a tail conjoining jamo, and have the result convert back into the precomposed syllable with the tail.

Theoretically, to look right all the jamo in the syllable should be able to change shape, size, and positioning in response to the other jamo in the syllable. That's difficult to implement at the font level.

One simple way to make it work is to sacrifice the layout. Suppose we split up the character box in a fixed, compromise way, into sections for the lead, vowel, and tail, and then define the glyphs for those things so that they will nicely fill their respective areas. Define the leads and tails to be zero-width characters, with the lead glyphs appearing to the right of their reference points and the tails to the left; and define the vowels to be full-width characters. Then if you typeset a lead, a vowel, and optionally a tail (bearing in mind that every syllable contains one lead, one vowel, and zero or one tails), with no kerning, the result is that the parts overlay one another and produce a sort of okay-looking syllable box. It won't be great because the spaces assigned to each jamo don't change depending on the other jamo in the character, so that for instance all the leads get shoved over to the left even when the vowel is something horizontal, because there could be a vertical vowel and there has to be space reserved for it on the right in every lead jamo glyph.

Conjoining jamo overlaid in this way look like 「지읍시다」; compare with precomposed 「지읍시다」. It does not look very appealing, especially because that's a near-worst-case example where three out of four syllables have empty tails and the fixed layout cannot stretch jamo to consume the resulting blank space; but this approach has the big advantage that it can express syllables full of archaic jamo combinations, such as 「ᄉᆑᇗᅙᅤᇫ」. Hundreds of thousands of syllables can be constructed that way, far exceeding any reasonable set of precomposed glyphs.

There are two layers of additional processing in place to improve the typesetting of conjoining jamo. First, OpenType substitution rules in the "ljmo" and "vjmo" features, which should be turned on by default (but must be activated manually in current XeTeX), recognize cases where a different layout would be better. The precomposed jamo vary the entire layout of each syllable depending on all the jamo in the syllable, so that any given jamo might appear in hundreds of subtly different sizes and locations. It is not practical to store all those variations of every jamo as individual glyphs, but Jieubsida compromises by recognizing six different standardized layouts for a syllable, depending on the shape of the vowel and the presence or absence of a nonempty tail.



The default forms of all the conjoining jamo, which appear if the substitution features are turned off, are the layout 0 forms. Some vowel jamo extend in both the vertical and horizontal directions, but in practice the most often-used ones are only vertical or only horizontal, and such vowels can be reused for layouts 1 and 2; presence of the relevant vowel is what triggers the switch to the other layout for the lead and tail. In addition to the layout 0/1/2 forms, each vowel jamo has an alternate form for tail-less syllables. Lead jamo have four alternate forms in addition to layout 0, since the forms for layouts 1 and 3 are identical and can share a glyph. An additional copy of the layout 0 leads also exists, as part of the workaround for a bug in XeTeX's handling of GSUB substitutions. Tail jamo do not need alternate forms, because they only appear in layouts 0, 1, and 2, and have the same space allocation in all three layouts. All these extra glyphs are coded into the private use Unicode code points

between U+FF200 and U+FF6FF, and can dangerously be turned off with the configuration code point controls, but it is not intended that they actually be accessed through the private-use code points; instead, the OpenType substitutions will put them in in place of the layout 0 forms from U+1100 through U+11FF, where necessary.

With the alternate forms chosen by the substitution rules, but no precomposed syllables, the font family name is rendered as 「지읍시다」. Note that is almost identical to the precomposed-syllable version, 「지읍시다」. There are some minor differences in the size and spacing of the jamo, because the glyph-overlay approach basically makes kerning impossible, and the six canned layouts still do not quite match the per-syllable customized layouts of the precomposed syllables.

The last layer of processing consists of ligature rules in the OpenType "liga" feature, turned on by default, which recognize jamo sequences that correspond to the precomposed syllables and substitute precomposed glyphs wherever possible. In order to work properly (in particular, for correct recognition of tail-less syllables) the alternate-layout substitutions must have already run before these ones. With the ligature substitutions in effect, actually entering the conjoining jamo for the standardized present-day Korean language will give the same result as using the precomposed code points. The family name entered this way appears as 「지읍시다」, which should be both visually and logically (same glyph sequence) identical to 「지읍시다」.

Also worth mentioning is the "ccmp" feature, which is supposed to run before all others. It splits precomposed syllables that have no tails into their component lead and vowel jamo code points, and joins consecutive conjoining jamo into clusters, where possible. If you are not doing anything weird with Unicode canonical equivalence, this is unlikely to have much effect; the split precomposed syllables will be recombined by "liga," and you won't have any clusterable conjoining jamo in the input anyway. However, this level of processing is needed (with the others) in order to do some of the strange things that the Unicode Consortium in its wisdom thinks you want to do, such as putting conjoining tail 「ㅁ」 after precomposed 「기」 and expecting it to turn into precomposed 「김」. I'm told that not many fonts actually support that. Jieubsida does! However, I cannot promise

that absolutely every weird Unicode thing will work. Most of them should.

The big win is that with all the substitution features enabled it is possible to mix common conjoining jamo, rare conjoining jamo, and precomposed syllable code points, and the result will be the best that is reasonably possible, automatically invoking precomposed syllables where possible and best-fit alternates otherwise. Here is a nonsense example demonstrating all the different layouts: 「춲지퀘읍슁시꺠쿓다쌊」

ハンジャ　　Hanja

The above discussion applies to the hangul phonetic script. Korean has traditionally also been written with Han characters (called "hanja" in Korean), and they are still used a little bit, in combination with the phonetic script—somewhat like the use of kanji in Japanese, but with the important difference that it is considered acceptable to write present-day Korean entirely in hangul without any hanja, whereas kanji are a necessity for present-day Japanese. The best information I have is that most Korean people today are actually able to read few to no hanja, but they remain in use as abbreviations in contexts like corporate names and newspaper headlines where exact comprehension is not terribly important.

The trouble is that Han characters as used in Japanese, Han characters as used in Korean, and Han characters as used in other languages that might use them (such as Vietnamese and various dialects of Chinese) are all incompatibly different despite being "unified" to overlapping subsets of the same Unicode code point range. For instance, the character 「神」 meaning "god" is written as shown, in present-day standard Japanese, and is very similar to that in both "simplified" and "traditional" Chinese. The equivalent hanja character (at the same code point, U+795E) has a left-side radical that looks like 示 instead of ネ. In fact, I've seen the Korean form of this character in some archaic and ceremonial contexts in Japan, so the switch to the newer form in Japanese was probably quite recent; but if you write the present-day Japanese form where you should write the Korean form, it will look wrong.

The current versions of the Jieubsida/Tsukurimashou fonts address this issue by simply not supporting hanja at all.

Supporting hanja would mean going through the entire set of kanji, finding all the differences, and creating separate versions to be the hanja or parameterizing the differences or otherwise dealing with them all, and that is beyond the current planned scope of the project. So is supporting Korean in the first place, actually; I don't speak it, hangul got added because it seemed interesting, it was likely to be a relatively large payoff for a relatively small amount of work, and (as has been demonstrated by experience with, among other things, FontForge hinting bugs) it provided a good testbed for some techniques that will later be applied to the Japanese side of the project.

It's easy to imagine that someone could build a font containing both the kanji glyphs from Tsukurimashou and the hangul glyphs from Jieubsida, and then try to use it to write hanja. I hope you will not attempt that, and the Tsukurimashou build system is designed not to build such fonts. There are too many fonts like that on the Net already as a result of people's attempts to build fonts for "all of Unicode" without really understanding the consequences of what they're doing. Obviously, in an open source system I can't stop people from creating such chimeras for their own use, but I don't want someone to create a Korean-language document with Japanese fake hanja in it, be asked "What is that horrible font you're using that gets the hanja shapes wrong?" and have them say "It's Jieubsiuda by Matthew Skala!" Please, if you're going to set kanji next to hangul and call them hanja, try to make clear to people who see the results that it was your idea to do that and not mine.

フォントの名前ついて
Regarding Font Names

First, I will use the term "font" to refer to an OTF or similar file with a complete glyph set (or as many glyphs as have been selected with configure). This will generally be a few thousand glyphs for the Tsukurimashou family and about 13 thousand for the Jieubsida family. Because METATYPE1 is limited to producing Postscript font files with at most 256 glyphs each, the system builds "subfonts" each corresponding to a "page" of the Unicode character set—that is, an aligned 256-code-point block. For instance, the range U+0000 to U+00FF is the page containing ASCII and Latin-1 characters.

The most authoritative form of the name of a font is the "Hamlog name," which is an ordered quadruple of Hamlog atoms. These names are manipulated by, for instance, the code in select.hl. The items in the quadruple are called the family, the style, the weight, and the spacing. An example Hamlog name might be "(tsukurimashou, bokukko, demibold, monospace)."

The family represents the broad category of the font. Generally, all fonts sharing the same family value will have basically the same glyph set and the same shapes for all glyphs, though some fine details will vary. If there are significant differences in shapes, they will be put in a separate family. At present the fully supported values for the family are "tsukurimashou" and "jieubsida," though there is also alpha code for an italic family called "tsuita" and some experimental (less than alpha status) code exists for a family to be called "blackletter_lolita."

The possible values for style will depend on the family. Styles generally correspond to sets of preset values for the adjustable parameters of a family's letter-drawing code. The style values currently allowed, per family, are as follows:

tsukurimashou kaku, maru, mincho, bokukko, anbiruteki, tenshinokami.

jieubsida dodum, batang, sunmoon.

tsuita soku, atama.

blackletter_lolita cosette.

The weight is not presently used; the only implemented value is "normal." The plan is that in future it will be allowed to take six different values ranging from "light" to "extrabold," and corresponding to different levels of boldness. Still to be determined is how this will interact with style values that imply something about boldness, such as "tenshinokami."

The spacing may be "monospace" or "proportional."

The Hamlog name is translated into a few other kinds of names, by a combination of Hamlog code in select.hl and Perl code in the Makefile. One important translated name is the "short name," which is a sequence of two, three, or (never occurring in the current version, but allowed in principle) four short ASCII tokens, which in different parts of the code may be separated by either hyphens or underscores. An example short name is "tsuku-mi-ps," corresponding to the Hamlog name "(tsukurimashou, mincho, normal, proportional)." Each element in the Hamlog name is translated into a token for the short name, according to the table given by the short_name/2 predicate in select.hl, but some Hamlog atoms (namely "normal" and "monospace") translate to the special value "l___da" (for "lambda," meaning empty string) and are removed, usually leaving fewer than four tokens in the result.

Subfonts also have short names, formed by adding an extra token at the end which is the two- or three-digit hexadecimal representation of the page number. For instance, "tsuku-mi-ps-00" might be the short name of a subfont. These are often used in constructing filenames for intermediate files during build.

Fragments of short names are also used to name source code files. Files following this scheme (not all source files do) have names consisting of one family token, a hyphen, and one other token, followed by the appropriate extension. Building a subfont involves creating a temporary driver file that includes all the relevant source files for that font's short name. For instance, to build "tsuku-mi-ps-00.pfb" the build system would create a driver file that loads "tsuku-mi.mp,"

“tsuku-ps.mp,” and “tsuku-00.mp,” in that order. Some of those files might then go and load other files outside this naming scheme.

In the case of the family token not being “tsuku,” the build system will use source files that match the subfont’s family token where those exist, and default to “tsuku” when such files don’t exist. For instance, to build “bll-co-01.pfb” (Blackletter Lolita Cosette, page 01) it might load “bll-co.mp” (which exists), but then load “tsuku-01.mp” because “bll-01.mp” doesn’t exist. Thus, the “tsukurimashou” family can be thought of as a generic ancestor from which any others inherit.

Fonts also have “long names,” which are created by a similar translation process. Long names are generally mixed case ASCII, and the elements are separated by spaces and may also contain internal spaces. As with short names, normal weight and monospace translate to nothing and are removed when forming the long name. An example long name might be “Tsukurimashou Tenshi no Kami PS.” There are also some places, notably in the filenames of final OTF fonts, where a long name is used with all the spaces and hyphens removed.

There are also “JK names,” which are constructed just like long names but with elements written in Japanese or Korean, and thus generally in non-ASCII Unicode characters. For Jieubsida fonts these are in Korean and have spaces between the elements; otherwise they are in Japanese and have no spaces. The main use of JK names is to create font metadata. An example Japanese name might be「作りましょう天使の髪PS」, corresponding to “Tsukurimashou Tenshi no Kami PS.” Since there is some information loss in transliteration to English, and many of the names come from Japanese and Korean words anyway, it may be best to think of the JK name as the truest name of the font for human purposes, even though from the code’s perspective both are derived from the authoritative Hamlog name. The source of the translation table for JK names is in txt/jnames.txt, which is processed by Perl to change the UTF-8 into a hexadecimal form that Hamlog can handle, the result going into hamlog/jnames.hl.

『作りましょう』を作りましょう!
Building Tsukurimashou

Please note that if you just unpack the Tsukurimashou distribution and look in the "otf" subdirectory, you will find some ready-made font files there. If you are content with them, then those are the only files you need; you can safely delete everything else in the package (save the PDF documentation files, if you want) and ignore this sub-section. These notes on building Tsukurimashou are only for expert users who want the greater control, wider style coverage, and intellectual challenge of doing custom compilation.

Assuming you wish to embark on the adventure of compiling Tsukurimashou, you will need at least the following:

- A reasonably standard Unix command-line environment. I use Arch Linux. Anything branded as "Unix" should work. MacOS X might work. Windows with Cygwin might work.
- A standard C-language tool chain (normally comes with Linux).
- Perl.
- GNU Make (non-GNU versions will not work).
- Metapost (this comes with most TEX/LATEX installations)
- A version of FontForge that actually works.

Other things that might also be useful include:

- XƎLATEX (needed for compiling the documentation).
- Expect (makes the build system work better—exactly why is explained in more detail in the warning you will get if you don't have it).
- The KANJIDIC2 database (needed for the kanji coverage chart and a planned future fine-grained character subset selection feature).

- A supported Prolog interpreter (not necessary, but if one is available then the build system will use it and run more efficiently).
- A multi-CPU computer (the build system will by default detect and use all your CPUs; since some stages of compilation require a lot of processing, it's nice to have several).
- A checksum program, preferably sha256sum (used for some subtle build-system optimizations).

If you have a current version of TeXLive, then you probably have Metapost and XeLaTeX already. Note that it does have to be fairly up-to-date. Tsukurimashou now bundles its own version of the public domain METATYPE1 code, so the dependency of previous versions on the mtype13 distribution of that has been removed from the list. In preparing METATYPE1 code for bundling I discovered a previously unknown dependency of previous versions on the t1asm package from t1utils; that has now been bundled as well.

I mentioned a requirement for a version of FontForge "that actually works." FontForge is plagued by many bugs and numerical instabilities in its spline geometry code, and stock, distributed "stable" versions tend to hang and/or segfault when they are used to compile Tsukurimashou. As of April 2012, the version distributed by Arch Linux x86_64 seems to work. In the recent past there've been bugs relevant to Tsukurimashou fixed in the interface to FreeType for rasterization; spline geometry; and hinting. I can't promise that using the latest patched version will be enough to keep FontForge from crashing; what I do myself is go in with gdb after each sufficiently annoying crash, find the line that is segfaulting, and try to fix it. I'm not sure that I have reported or recorded all the changes I've made as a result of this procedure. Because of all this, Tsukurimashou's configure script will check that your FontForge has been compiled with debugging symbols, and generate a warning message if not. You can either get a debuggable version, or disable the message if you feel like living dangerously.

ブリドのシステム　　Build System

The build system is based on GNU Autotools and should be reasonably familiar to anyone who has compiled popular free software before. Run "./configure --help" for a description of available configuration options; run "./configure" possibly with other options as described in the help message to set up the system the way you want it; then once it is configured to your liking, run "make" to build it. The completed font files end up in a directory called "otf" in the distribution tree. A few ready-made ones should be there already when you unpack the distribution, for the benefit of the probable majority of users who can't do their own compilation.

If you run "make install" it will attempt to install the fonts and documentation files in sensible locations, but it's not really customary for fonts to be installed like other software, and you may be better off simply taking the compiled files from the otf directory and doing with them whatever you would normally do with fonts instead of using the automated install. At this time there is no TEX-specific font installation support, though that might be a nice feature for me to build in the future. The install target uses the "--prefix" and similar options to configure in a reasonably obvious and standard way.

If you have KANJIDIC2, place it (in the gzipped XML form in which it is distributed) in the build directory, the doc subdirectory off of the build directory, your system's dict or share/dict directories, possibly under /usr or /usr/local or your configured installation prefix, or your home directory or "~/dict"; or put it somewhere else and tell configure where with the "--with-kanjidic2" option. Although what's used by Tsukurimashou is factual information not subject to copyright, and KANJIDIC2 (which also includes creative content that might be subject to copyright) is distributed freely, it is not distributed under a GPL-compatible license and so for greater certainty I'm avoiding including KANJIDIC2 or any data extracted from it in the Tsukurimashou distribution. What you get if you have KANJIDIC2 is any features that depend on knowing the grade levels and similar properties of kanji characters—such as the chart showing how much of each grade has been covered, which is important for

my own development efforts in planning what to design next and knowing when to release.

The build system for Tsukurimashou is fairly elaborate; it may seem like overkill, but given that I expect to run this myself several times a day for at least a couple years, it's important that it should be pleasant for me and efficient in time consumption. Thus it defaults to "silent" build rules, uses pretty ANSI colours, and does a bunch of complicated checks on whether file contents (not just modification times) have changed in order to avoid expensive dependency rebuilds unless those are really needed, while triggering them automatically when I add new source files.

One important tip is that if things are failing, you can add "V=1" to the "make" command line, as in "make V=1 pfbtmp/tsuku-kg-00.pfb" to temporarily disable the silent build rules and see what's going on. Note no hyphen, because this is a variable setting instead of an option, and you will probably also want to specify a target filename, which implicitly overrides the default "-j" multi-CPU option. In version 0.6, Expect was rendered less necessary by the replacement of mtype13 with a homegrown and less chatty Perl script; but the support remains in place to handle TEX's lesser jubilations.

You can also add "KEEPTMP=1" to prevent deletion of the temporary directories created while running Metapost. This feature is primarily useful for debugging the pfb-generating scripts themselves.

You can turn subsets of the character set on and off with the "--enable-chars" option to configure. Give it a parameter consisting of a comma-separated list of tags with optional plusses and minuses. The idea is that each tag represents a subset of the character set, and they are evaluated in the order specified. The default is "all," meaning all characters defined in the source code will be included in the fonts. Other tags currently supported are "none" (which actually has no effect but is included for readability purposes), "ascii," "latin1," "mes1," and families of tags named like "page12," "uni1234," and "u123ab," with lowercase hexadecimal digits, which correspond to 256-character blocks of Unicode code points and to individual code points. For example, the specifier "all,-page02" would include every character it

can except none in the range U+0200–U+02FF; the specifier "uni0041" would create very small fonts containing only the uppercase A from the ASCII range. The default is "all." The point of this system is to make it easier to generate stripped-down fonts for Web embedding and to reduce compile time during maintenance, when I may want to focus on just one subrange of the character space for a while.

The "--enable-ot-features" option works the same way for OpenType features; note that OpenType features may also be automatically and silently disabled in whole or in part, overriding this setting, if you have used the previous setting to disable characters necessary to implement the features. For instance, you can't have OpenType contextual substitution support of fractions or enclosed numerals if you have disabled the numeral characters—though you *can* build a font with enclosed and not regular numeric glyphs, because glyphs are mostly independent of each other.[1] This automatic disabling may not be perfectly clean, either; in some cases disabling a character might not disable the feature code that mentions it, and in that case FontForge may create an empty glyph for the character even though you disabled it. That shouldn't happen unless you attempt something very unusual and non-standard, however. Currently supported tags are "all," "none," and a four-character tag corresponding to the OpenType name of each feature that exists: "afrc" (alternate, that is, vertical or nut, fractions), "ornm" (ornaments), "smcp" (small caps), "ss01" (stylistic set one, heavy metal umlaut), and "ss02" (stylistic set two, enclosed characters).

The configure script is designed to accept an option for a similar plus/minus selection of font styles, but the code in other places to support that option does not exist yet and if specified it will have no effect on what really gets built.

The "make dist" target defaults to building a ZIP file only, instead of GNU's recommended tar-gzip. This decision was made in order to be friendlier to Windows users, who tend to have seizures when

[1]Mostly. In the case of white-on-black reversed glyphs and some fractions precomposed by FontForge instead of by METATYPE1, you must include all the parts that FontForge will assemble in order to get the combined glyph made by assembling those parts. This is a sufficiently arcane scenario that the build system will not check for it.

confronted with other archive formats, and TEX users, who are accustomed to ZIP as well. However, the makefiles should also support many other formats via "make dist-gzip" and similar.

I am not confident that "make dist" will really include everything it should if you have disabled optional features with the above options to configure. Please "make me one with everything," as the Buddhist master said at the hamburger stand, before trying to build a distribution.

The "make clean" target and its variations probably do not really make things as clean as they should.

Don't bother with "--disable-dependency-tracking"; that is an Autotools thing meant for much larger and more softwary packages. It applies only to code in languages supported by Autotools, which at present means only the C kerning program whose dependency is trivial. The dependency tracking for METATYPE1 code is completely separate, unaffected by this option, and trying to disable it would be a bad idea.

Autotools encourages the use of a separate build directory, with the sources remaining inviolate elsewhere, but that is not really recommended for Tsukurimashou. I try to make it pass "make dist-check" right before each release, which implies making separate build directories work, but if you are building Tsukurimashou from a checked-out SVN version, chances of that working are slim. It's safer to build right in the main directory. Even if VPATH builds work, they are only intended for the case of having an untouched set of sources in one directory and a build in another. If you try to do the overlay thing, with modified versions of some source files in your build directory, it is unlikely to work, because of the large amount of bolted-on filename and path logic that doesn't go through GNU Make for name resolution.

If you look in the source of the build system, specifically files like configure.ac, you'll see that I did a whole lot of work ripping out sections of Autotools that were designed for installations of executable software. GNU standards require the definition of a ridiculous number of different installation directories, almost none of which are

applicable to a package of this type, and I took out most of the support for those to reduce the cognitive load for users who would otherwise have to think about their inapplicability. This package doesn't install any executables, libraries, C header files, or similar, at all. Cross-compilation and executable name munging were removed for the same reason; a C program does get compiled to do the kerning, but it is only meant to run on the host system during build, with all the installable files being architecture-independent. The hacking I did on Autotools means that if you modify the build system such that you would be re-running Autotools, it's likely to break unless your version of Autotools is close to the 2.65 version I used. The configure script will try to detect such a situation and warn you.

The design of this build system was influenced by Peter Miller's interesting article "Recursive Make Considered Harmful," available at http://miller.emu.id.au/pmiller/books/rmch/.

ツールの文書化　　Tool documentation

There are a number of scripts included in the tools/ subdirectory. Most of these are intended only to be invoked automatically by the build system, but two intended to be invoked by hand are described below.

- autodep: automatically maintain the header inclusion lines in METAFONT files, so that each file will include the files that define macros it uses, and only those. Files intended for use with autodep should include a comment line consisting of the case-sensitive text "% AUTODEPS" and any automatically-maintained input lines following that, terminating with a blank line. When the tool is invoked with the command line "tools/autodep mp/mp/*.mp" from the root of the source tree, it will scan all the METAFONT-language sources, find definitions of macros, and where necessary rewrite the blocks of includes tagged with the special comment line in order to ensure that every file includes the definitions of all needed macros. There are a number of exceptions and special cases included to make it do the write thing with macros defined in multiple places as a result of style

overrides, and so on.

- udcpyright: check that all years in which a file was modified are included in its copyright notice. This assumes one has an SVN checkout in the current directory—intended to be a checkout from my private SVN repository, but it would probably work with one from the SourceForge repository as well. It scans all SVN-controlled files for anything that looks like the project's standard copyright notice, and then reports on any files for which the set of years mentioned in the notice does not match the set of years in which SVN has log entries for that file. The idea is that once a modified version of the file has been checked into SVN in a given year, it has been "published" in that year and the year should be mentioned in the copyright notice; but when a new year comes we don't want to just go through and modify all files solely to insert another year in the copyright notice. By running this every so often, and manually updating the notices when appropriate, the hundreds of copyright notices in the project should painlessly converge on correctly reflecting the years in which files were actually changed.

For completeness, here is a summary of the functions of all the other scripts in the tools/ subdirectory. Most of them are written in Perl.

- add-flog: attach the "font log" file to a FontForge SFD-format font; this requires re-encoding it to UTF-7 format.
- livecode: chase file inclusions and determine all the "live" (that is, actually invoked) macro defintions in a list of METAFONT-language files. The output of this is not (because lines may appear in the wrong order) actually code one could run; but it is intended to have the property that it will change if and only if changes in the input necessitate recompilation. If you modify a macro that is not actually called, livecode's output should not change because it won't include that macro. Then the build system can use a checksum of livecode's output for fine-grained change detection, to skip recompiling some files that (under

GNU Make's timestamp-based scheme) would otherwise appear to require recompilation.

- `make-ass`: generate the `assemble-font.pe` script, which in turn does most of the work of assembling 256-glyph subfonts to create full-coverage OTFs.

- `make-book`: make decisions on how to split the multi-thousand-page proof document into smaller "books." The output is a set of TEX files which then get processed further to generate the actual PDFs.

- `make-cfghl`: translate `configure` command-line strings into Hamlog for inclusion in `config.hl`, which then feeds back to control the build system.

- `make-eids`: scan the proof files and generates an EIDS-format dictionary of character decompositions.

- `make-fea`: perform text substitutions on "feature source code" (fsc) files to generate Adobe-format "feature" (fea) files expressing code-like font metadata.

- `make-flog`: collect a bunch of debugging information and generate a font log file (see `add-flog` above).

- `make-gpos`: compute information for GPOS features "mark" and "mkmk" by scanning the proof files (to determine anchor locations and compatibility) and some generated FontForge scripts (to determine bearing adjustments made during kerning).

- `make-hglpages`: generate METAFONT source code for the hangul precomposed syllable subfonts, which are all identical except for their starting indices and generated from a template.

- `make-kchart`: generate TEX code for the kanji coverage chart.

- `make-kddata`: extract KANJIDIC2 data in Hamlog form.

- `make-name`: generate a font's NAME metadata table.

- make-proof: generate TEX code for proofs and pretty-printed source code.
- mp2pf: stripped-down translation from Awk to Perl of a similar program in METATYPE1; this invokes Metapost in the appropriate way to generate something that can be processed into a Postscript font, and then does process the result into a Postscript font. Note that hinting support has been *removed* from this code, and the resulting Postscript fonts may not really be good or entirely format-valid; they are only intended as a way of getting the outline data into FontForge for later stages of processing.
- mpdep: generates a list of the dependencies of a METAFONT-language file (or set of them), for automated Makefile generation. Compare with autodep, which actually edits the files to include the other files they should based on the macros they use; mpdep only looks at what other files each file does include, and reports that.
- progress: invoked during build to give progress reports and completion-time estimates.

ブリドのシステムの図示　　Build system diagrams

This section is still under construction, but it is planned to be a graphical summary of what is going on in Tsukurimashou's rather complicated Makefile. Figure 3 describes the symbols used, and Figure 4 shows the process by which fonts are built. Proportional fonts go through additional processing to generate the kern tables and so on, as shown in Figure 5.

Be aware that these diagrams are meant to clarify, not to formally describe, the code; this is *not* a formal modelling language like UML, and the semantics of the symbols are not necessarily entirely consistent. Also left out of the diagrams are language interpreters like Perl, and a great deal of information flow to and from the main Makefile itself. It passes configuration information, mostly in command-line arguments, to pretty much all parts of the system.

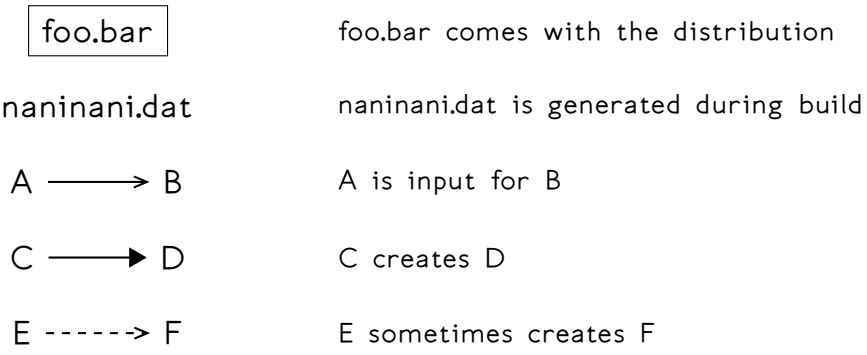


Figure 3: Legend for build system diagrams

ハムログ　　Hamlog

Evaluating the consequences of build options like "--enable-chars" requires doing a certain amount of logical inference for which shell scripts are not well-suited. It might be possible to get GNU Make to do the necessary computations, inasmuch as it's quite programmable, already required to build Tsukurimashou, and fundamentally a logical inference engine at heart. But that would probably involve creating many tiny files corresponding to logical propositions, which would waste space and cause other problems on some filesystems. A more elegant approach would be to use a real logic programming system, i.e. Prolog—which happens to be one of my research interests. But I didn't want to create a dependency on a Prolog interpreter because I think users will object to that; the existing dependencies of this package are already hard enough to sell. I also didn't want to bundle a Prolog interpreter, even though good ones are available on permissive licensing terms, because of the file size and build-system complexity consequences of bringing a bunch of compiled-language software into the Tsukurimashou distribution.

The solution: Tsukurimasho's build system will look for Prolog, and use it if possible. But the package also ships with something called Hamlog, which is a toy Prolog-like logic programming system written in Perl. (A ham is like a pro, but less so.) If the build system can't find a Prolog interpreter, it will use Hamlog instead. Hamlog is slow, and internally horrifying, but it works in this particular application. It is not particularly recommended for any other applications.

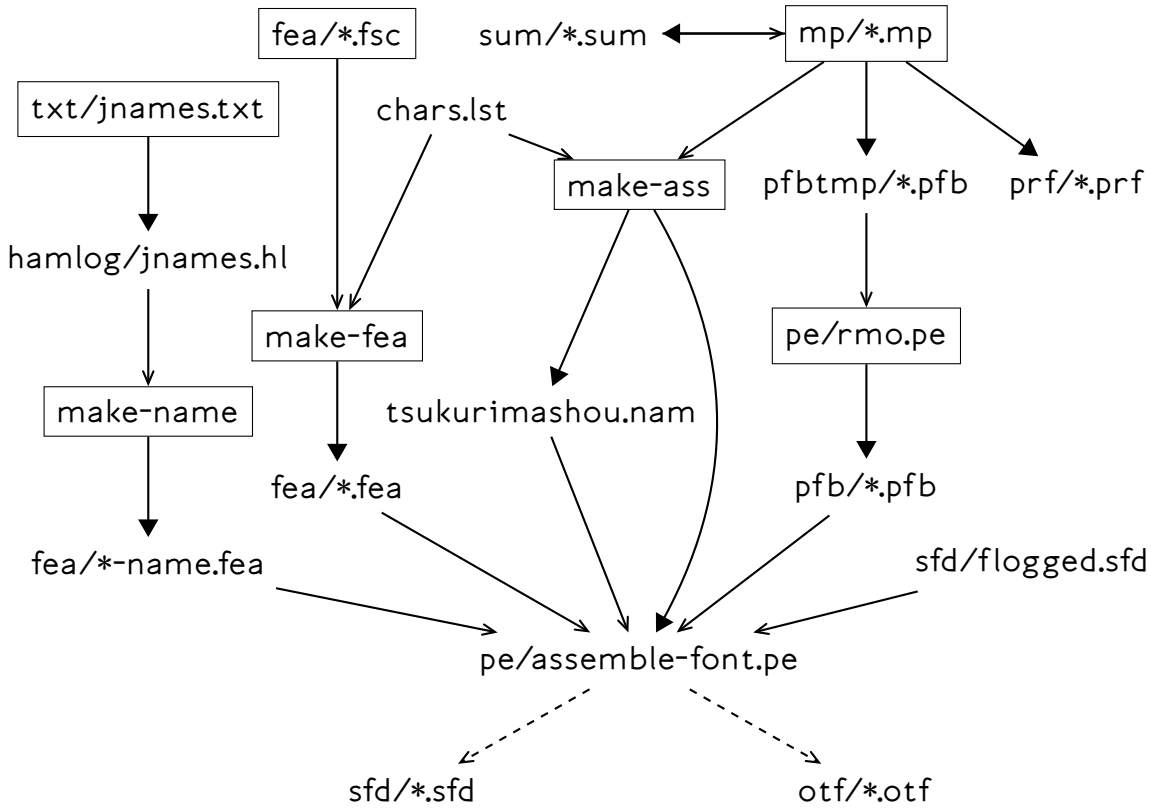


Figure 4: Building the fonts themselves. Checksums in sum/ provide fine-grained change tracking on the Metapost files in mp/, which define Postscript "page" fonts each covering up to 256 code points. Those are processed to remove overlaps (RMO) and end up in pfb/. The make-ass script also scans the Metapost files to get a list of defined code points and some other metadata; its main output is a FontForge script that assembles the page fonts into full-coverage OpenType or FontForge native fonts. Non-proportional fonts become OpenType and are finished here; proportional fonts become FontForge native and are processed further. The assemble-font script also makes use of per-family feature files from make-fea (defining things like glyph composition) and per-font feature files from make-name (defining metadata table contents). The "proof" files in prf/ are generated as a side effect of compiling the PostScript page fonts, and pass information about glyph construction to both the automatic documentation generator and the GPOS table generator.

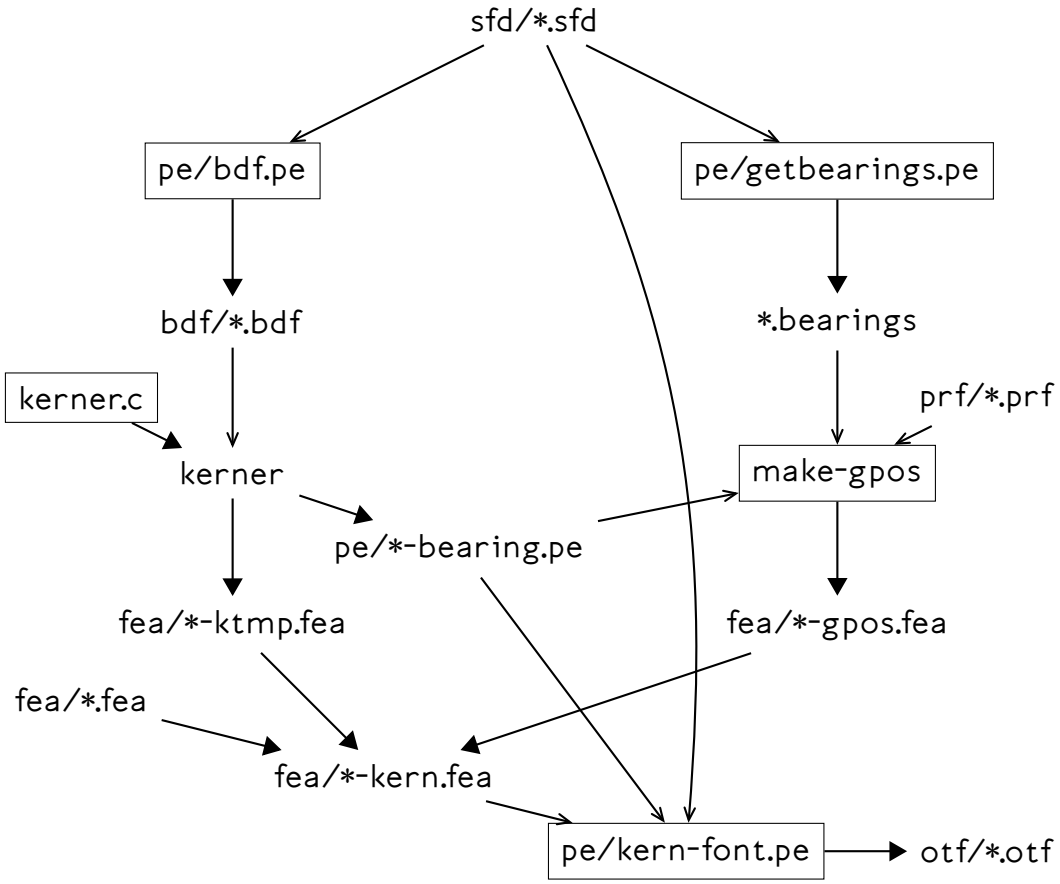


Figure 5: Additional processing for proportional fonts. The SFD file containing the font outlines is converted to a BDF bitmap font, which feeds the kerner program. That generates a "ktmp" feature file with the actual kerning data, and a FontForge script to apply the chosen bearings. At right, the make-gpos script reads the proof files to determine accent anchor locations. But since those must be adjusted when the bearings are set, it also needs "before" bearing information extracted by getbearings, and "after" obtained by reading the bearings script from kerner. The feature files for kerning, mark composition (GPOS), and family features are all combined and applied by the kern-font script, which runs the bearings script as a subroutine, finally producing a complete OTF font.

The configure script looks for SWI-Prolog, ECL^{i}PS^e-CLP, and GNU Prolog. If one of these can be safely detected, it will be used. ECL^{i}PS^e-CLP has the issue that it shares a name with a widely-used programmer's IDE, so it is not safe for the configure script to actually execute an executable called "eclipse" if it finds one. If something that might be ECL^{i}PS^e-CLP is detected, the configure script puts up a warning and the user is free to enable it explicitly.

The rest of this subsection can and probably should be skipped by anyone who isn't both a Perl and a Prolog hacker.

Still with me? The way Hamlog works is sort of fun and so I'm going to spend a few pages describing it for those who are interested, if only as an example of something you should Not Try At Home. The idea is to use regular expressions for the operation of finding a clause whose head matches the current goal. Hamlog reads its program into a Perl hash, where the key is the functor and arity (like "foo/3") and the value is a newline-separated pile of clauses in more or less plain text. When it tries to satisfy a goal, it takes the goal (which starts out as plain text) and converts it to a regular expression that will match any appropriate lines in the database. Variables in the goal turn into wildcard regular expressions; ground terms turn into literal text; and then when there's a match, parenthesized sub-expressions are used to extract the body of that clause.

It is because of the use of regular expressions that Hamlog doesn't do compound terms, and in turn is likely not Turing-complete (though I haven't thought carefully through all the possibilities of using recursive predicates to build data structures on the stack). As all the world knows, it is impossible to write a regular expression to match balanced parentheses. Current versions of Perl actually bend the theory with experimental extensions that do allow the matching of balanced parentheses, so that in a certain important sense Perl regular expressions *are not regular expressions anymore at all*, but even I am not quite twisted enough to actually deploy such code. Things in Hamlog that look like compound terms (such as the subgoals in a clause body) are handled as special cases; but the point is that arguments to a functor that will be used as a goal have to be either atoms or variables. This also means Hamlog doesn't do

Prolog syntactic sugar things that expand to compound terms, such as square brackets for lists.

Once there's a match, it does string substitution on the matching head, the current partially-completed goal, and the body, to get a new modified body for the clause, taking into account any variables assigned by the head match. The new clause body gets substituted into the current partially-completed goal (which is a string) as a replacement for the head that just matched. So the partially-completed goal is a sort of stack of comma-separated heads that grows from right to left and implicitly contains all the assigned variables.

Because of the simplistic way variables are given their values, it is dangerous to use the same variable more than once in the same head, so constructions like "foo(X,Y,X)" should not be used. If you want to do that you should instead write "foo(X,Y,Z):- =(X,Z)." Note the non-sugary use of "=" as a functor, since the more common infix notation isn't supported. Note also that there should be a space between ":-" and "="; Hamlog doesn't require that but it may reduce the likelihood of parsing problems should the same code be run on interpreters other than Hamlog.

Variables in clause bodies are renamed once (using the clause serial number), when the clauses are loaded; as a result if the same clause body gets expanded a second time while variables from its earlier expansion are still unassigned, there could be trouble. This is not a very likely scenario, but it's worth mentioning.

Clauses in the database have serial numbers; and when a choice point goes on the stack, the serial number of the clause at which it matched is part of the record on the stack. Then if the interpreter backtracks to re-satisfy a clause, it writes the regular expression in such a way that it can match all and only serial numbers greater than the last place it matched. Creating an "integer greater than N" regular expression was surprisingly difficult—it's a simple enough concept but there are several cases that all must be handled properly or weird bugs turn up.

Syntax is simplified from Prolog. Variables start with an uppercase letter or an underscore and may contain uppercase alphanumerics

and underscores. Atoms start with a lowercase letter or numeric digit and may contain lowercase alphanumerics and underscores. For Prolog compatibility, atoms starting with digits should not contain anything other than digits, and the only atom starting with zero that should be used is zero itself; but Hamlog doesn't care about those points. Things containing a mixture of upper- and lowercase alphabetic characters should not be used. The special tokens "!" and "=" are technically treated as atoms too, but you should only use them in their typical meanings of cut and unification, and "=" should only be used with the general prefix syntax applicable to all functors, not as an infix operator (see above).

Variable names starting with, and especially the unique variable name consisting entirely of, an underscore, are not special in Hamlog. Beware, that means "foo(_,_)" contains only one variable occurring twice, not two distinct variables as it would in Prolog, and it violates the "only one appearance of each variable in a head" guideline. The unique variable name consisting of one underscore is probably best avoided entirely. But it may be desirable to use variable names starting with underscores anyway in some cases, because of their specialness to Prolog interpreters. I was really tempted to allow and use arbitrary UTF-8 (in particular, kanji) in atoms but refrained because of the desire for Hamlog code to be easily readable by nearly all Prolog interpreters.

I tried to keep the number of built-in predicates to an absolute minimum, partly because any that are not standard Prolog have to be re-implemented in Prolog (and probably once for every supported Prolog) to build the shell that executes Hamlog programs on a Prolog interpreter. Here's an exhaustive list.

- ! [cut]. This is implemented by string substitution as well: when a clause body gets added to the to-satisfy stack, there's an additional regular expression substitution pass that converts any instances of !/0 in the body into !/1 where the argument is an integer identifying the current height of the choice-point stack. If at any point in the future we attempt to satisfy a !/1 goal, then the stack gets popped back to that point (discarding

any choice points created between the time the ! got its argument and the present time). For this reason, ! should not be used as an atom or functor for any other purposes than as the cut, even if to do so would otherwise be valid Prolog.

- fail, which causes immediate backtrack (useful in conjunction with cut to implement negation).

- true, which is not actually used, so maybe I should delete it.

- var, true if the argument is a variable (not yet bound to an atom). This is important in Hamlog because many predicates need to be written to accept more than one instantiation pattern for their arguments.

- atom, true if the argument is an atom. This is implemented by rewriting the database entry for atom/1 on the fly; when you call "atom(foo)" it magically changes to having been defined as either the single fact "atom(foo)." or nothing, depending on whether "foo" is an atom.

- atom_final(AZ,A,Z), where AZ is an atom whose last character is Z and A is everything except the last character. Used for building and tearing apart atoms like "page00." This requires some careful handling in other interpreters because Hamlog has no concept of quotation marks and treats single-digit integers exactly the same as atoms whose names are the ASCII digits; real Prologs have more subtle type handling. As with atom/1, this is implemented by magically creating appropriate clauses on the fly.

- =/2. This also creates clauses on the fly. At least one of the two arguments should already be atomic when the goal executes, though that is rarely difficult to guarantee in practice.

And that's it for built-in predicates. Note that goals in a clause body also can only be combined with comma for conjunction (no semicolons for disjunction, and without them parentheses become unnecessary and are not supported either). There is also no syntactic support

for negation. However, you can (and the existing code does) compute negation and disjunction using multi-clause predicates, cut, and fail. What you can't build is any kind of I/O—so how can Hamlog programs communicate with the outside world?

The interpreter (after checking for the "--version" and "--debug" options, which do fairly obvious things) interprets its first two command-line arguments as a template and a query. The template ought to be a valid Prolog compound term for compatibility with other interpreters, but Hamlog actually treats it as a string. Then it backtracks through all solutions to the query, attempting to instantiate all variables, and writes (newline separated to standard out) all the *distinct* values assumed by the template. This is basically the same operation as Prolog findall/3 followed by sort/2, which is how the Prolog shell for Hamlog implements it. Any remaining command-line arguments, and standard input if there are none of those, will be read in the usual <> way to fill the Hamlog program database. Hamlog code is conventionally stored in "*.hl" files.

In the build system, the string of comma-separated tags for things like characters to be selected gets converted (by the "make-cfghl" Perl script) into a few clauses of Hamlog and written into the file config.hl. Also written there is a list of page_exists/1 facts naming the 256-code-point blocks for source files that exist in the mp directory. Then elsewhere in the build system, it invokes Hamlog with appropriate queries against config.hl and select.hl to get lists of characters, OpenType features, and other things that the user does or doesn't want, based on knowledge built into select.hl of what the different selection tags actually mean.

It is planned that in a future version, the KANJIDIC2 file will be automatically translated to more Hamlog facts expressing which kanji are or aren't included in the different grade levels; then it will be possible to use options like "--enable-chars=kanji,-grade3' for finer-grained selection of kanji.

In the case of a Prolog interpreter other than Hamlog, there is some other code written in that interpreter's own language to allow it to execute Hamlog programs and export something resembling this

command-line interface to the Makefiles. Since Hamlog programs are also syntactically valid Prolog, this support shouldn't be difficult in general. See the swi-ham.pl file for what currently exists of this nature. The main advantage of using a non-Hamlog interpreter is simply speed.

カーニングしかた
Kerning

This is a summary of how the automated proportional spacing and kerning code works.

First, the build system generates the future PS font as an OpenType file, all complete except for widths and kerning. It then calls bdf.pe, which sets all the bearings to 50 font units and makes a BDF-format bitmap font scaled so that the reference kanji square (1000 font units) takes up 100 pixels. This script filters out glyphs that should not be subject to kerning, which includes those with zero width (mostly combining characters), all the hangul individual-jamo code points (whose layout is handled by OpenType substitution features, and kerning would just make it too complicated), and the special glyphs used by vertical fraction composition (much the same situation as the hangul jamo).

The "kerner" C program reads that BDF font, puts all the glyphs into a common bounding box big enough to contain any of them, and finds the left and right contours—basically, the x-coordinates of the leftmost and rightmost black pixels on each row—for each glyph, as well as the margins, which are defined as the x-coordinates of the leftmost and rightmost pixels on any row of the glyph.

There is some special processing applied to the contours to make them more suitable. Consider what happens in a glyph like "=": many horizontal rows, namely those above and below the entire glyph and those in between the two lines, contain no black pixels at all, and so the leftmost and rightmost black pixels in those rows are formally undefined. If we set it next to another glyph like "." which only has ink in rows where "=" does not, then a kerning algorithm that looked only horizontally might let the period slide all the way under the equals and out on the other side. There has to be some vertical effect to prevent that. So the Tsukurimashou kerning program makes a couple of passes over each contour (one forward, one backward) to enforce the following rules, which (except for glyphs containing no black pixels at all, which are removed from consideration) fully define where the contour should be.

- The right contour cannot be any further left than the rightmost black pixel in the row.
- The right contour cannot be more than 10 font units (one pixel) further left than its value in the next or the previous row.
- If the right contour in the next or previous row is left of the *left* margin, then the right contour in this row cannot be more than 3 font units further left than in the next or previous row.
- Subject to the above rules, the right contour is as far left as possible.
- The left contour's rules are the mirror image of these.

These rules can be imagined as simulating something like the way letterpress printers kern type by physically cutting the metal type bodies at an angle to fit them more tightly together: it's bad to cut off part of the actual printing surface; you basically cut at a 45° angle; but with glyphs that only have a small vertical extent, so that the 45° angle would cut all the way across to the other side, then you want to use a more vertical angle (in this case, 16.7° from vertical) so you don't end up setting the next character actually earlier on the line. Figure 6 shows a typical example of the contour computation.

The right margin is subtracted from the right contours and the left margin from the left contours to get, for each glyph, a vector of numbers describing its shape along the left and right sides, independent of the width of the glyph.

All the left contour vectors, and (independently) all the right contour vectors, are subjected to a modified k-means classification. Initially, the contours are put in 200 classes, according to the values of a simple hash function applied to the glyph names. This is a change from earlier releases in which a round-robin was used: the advantage of the hash function is that although it remains deterministic, it helps break up a phenomenon that tended to happen with the Genjimon glyphs, where because of excessive symmetry in the initial arrangement, the classifier could never put the similar glyphs together.

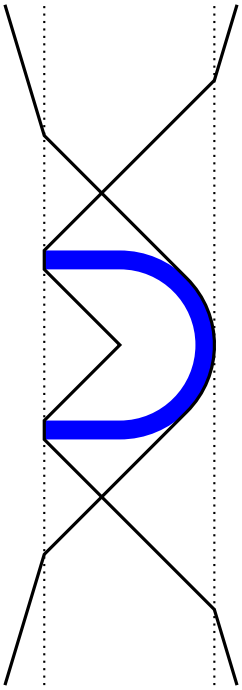


Figure 6: Glyph contour computation

Then for each contour the program asks the question "How far is this contour from the centroid of its class, and if I moved it to a different class, how far (after accounting for the fact that the centroid changes when I add the glyph) would it be from the centroid of the new class?" If moving the glyph to some other class would make it closer to the new centroid, then the glyph gets moved to the other class where its distance to the centroid will be minimized. Note that a glyph in a class by itself will never want to move out of that class, because its distance to the centroid is already zero.

There is an extra rule that the classification will never move a glyph into a class in such a way as to make a class larger than the "class limit," defined to be the larger of 100 glyphs, and three times the total number of glyphs divided by the total number of classes. The purpose of this rule is to counteract a tendency seen in some experiments for the classifier to create a few huge classes (for instance, a single class containing a large fraction of the 11172 precomposed Korean syllables) that cause font subtable size problems. It is not clear just where the limits are on how big a class may be, but this limit appears to work at the moment.

Glyphs are examined in this way until no more such moves are possible. The idea is that at the end of it, the glyphs will all be in classes that are as tightly clustered as possible. It's not guaranteed to be a global optimum (in other words, it's possible that some other assignment of glyphs to classes might be better; really optimizing this problem is difficult) but it's guaranteed to be a local optimum in the sense that it can't be improved by changing the assignment of just one glyph, and it's expected to be pretty good overall. Note that the initial assignment was deterministic (where random would be more usual for this kind of algorithm) because it seems undesirable for the kerning distances, overall, to be non-deterministic; my copy of the font shouldn't have different metrics from yours if they were compiled from the same sources with the same options.

After classification, we've got 200 classes of left contours and 200 classes of right contours. The actual kerning is done class-to-class, using the class centroids, rather than glyph-to-glyph. That way we will end up with up to 40000 kern pairs instead of millions. OpenType supports this kind of kerning pretty well. There will be a feature file generated listing the contents of the classes and the distances for each pair of classes, and that's much more efficient both in source and compiled form than specifying a distance for every pair of glyphs.

The number 200 (up from 150 in earlier versions) was chosen by educated trial and error. Every glyph must be in a class for full kerning; but no class can be too big or FontForge barfs. That in turn limits the size of the average class to some maximum, and so limits the number of classes to some minimum. Using 200 classes seems to be enough for the current Jieubsida fonts, which are roughly 13000 glyphs each, and it is reasonable to estimate that none of the fonts created in the planned scope of this project will have much more than that many glyphs.

To kern two contours together, we can compute a closeness value for each row by saying "if we positioned the margins of the glyphs this much apart, how far apart would the contours be in this row?" That distance, divided by a constant representing the optimal distance (currently 230 font units) and raised to a power representing how much extra weight to give to the closest points (currently 3),

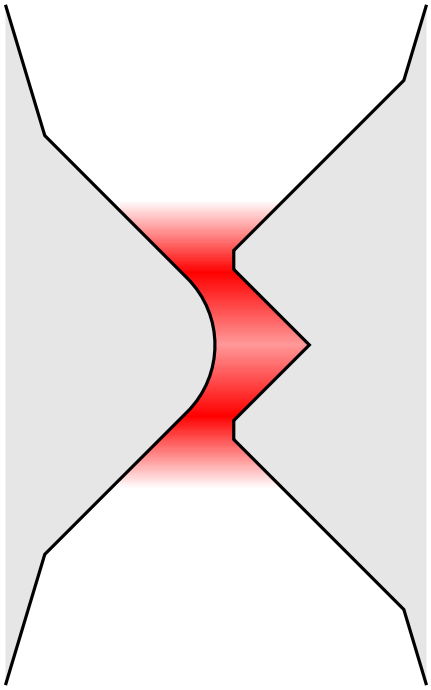


Figure 7: Closeness computation

represents closeness for the row. The sum of closeness for all the rows would be equal to the number of rows in the case of two perfectly vertical lines 230 units apart. The kerner program adjusts the margin-to-margin distance so that the sum of closeness is equal to that. It uses a binary search to do that adjustment, which is probably not optimal for a fixed exponent (there should be an exact analytic solution possible without iterating) but has the big advantage of not requiring a redesign should the exponent or even the entire closeness function change.

The closeness computation is shown schematically in Figure 7. Note that this closeness is calculated on the contours, as defined previously, rather than the actual shapes of the glyphs; it is also done on the centroids of the kerning groups, thus generic contours each representing many glyphs, rather than the contours of any specific individual glyphs.

The effect of the exponent 3 in the calculation is to give much more weight to points that are close together, as suggested by the shading in the figure. If we're kerning a pair like "]<", we want to

pay more attention to the point of the less-than than to the distant ends. An exponent of 3 means that points at half the distance count eight times as much toward overall closeness, so there's a strong bias toward seeing the points of closest approach. If we imagined using a larger exponent, this bias would be even stronger; in the limit, with an infinite exponent, the kerning would be determined solely by setting the closest approach to the optimum without reference to any other points. That is how most auto-kerning software works; but the results tend not to be good because in a serif font with a pair like "AV," inserting the ideal vertical-to-vertical distance between the serifs is going to place the stems, which are much more visually important, too far apart. Using an exponent somewhat less than infinity causes the stems to still have some significant weight. The value 3 was chosen by trial and error and may be subject to further adjustment.

Once all the class-to-class distances have been chosen, it remains to choose the bearings for the characters. Recall that kerning was computed from margin to margin, that is the amount of space to insert between the strict bounding boxes of the glyphs. Adding a certain amount of extra space to the glyphs themselves, and subtracting it from the kern distances, may result in a better, more concise kern table, as well as better results when glyphs from this font are set next to spaces, things from other fonts, and so on.

The first step is that the kerner program finds the average of all kern table values, and puts half that much bearing space on either side of every glyph, adjusting the kern values accordingly. This has the effect of giving every glyph an "average" amount of space, and changing the overall average kern adjustment to zero. If we were to throw away the kerning table at this point and just use the bearings, then in some sense these bearings would give the best possible approximation of the discarded kerning table.

Then for each left-class (which is actually a class of *right* contours: it is a class of glyphs that can appear on the left in a kerning pair) the program finds the maximum, that is farthest apart, amount of kerning between that left-class and any right-class. Two thirds of that kerning amount are added to the right-bearing of the left-class. The

concept here is that we generally want kerning to be pushing things together, not pulling them apart, so the "default" amount of kerning indicated by the bearing should be near the maximum distance apart, from which individual entries can then push things closer. Also, we generally want most (two thirds) of this adjustment to happen to the right bearing of the left-hand glyph in the pair.

Then to clean up the rest, the program examines each right-class (which is a class of left contours) and similarly finds the furthest-apart kern pair and adds that to the left bearing of the right class, adjusting all kern pairs appropriately. At this stage it's guaranteed that all the kern table entries will be zero or negative: kerning only pushes glyphs together from where they would otherwise be, it never pulls them apart.

Kern table entries are dropped if they are less than ten font units; that cuts the size of the table considerably. In a change from earlier versions, the kern values are not otherwise rounded (beyond being integers). The table is written to a fragment of an Adobe .fea file, with subtable breaks (on left-class boundaries) each time a subtable grows past 5000 entries; that means subtables actually end up a little over 5000 entries each. That seems to be how big they can get without overflowing the OTF table-size limits. Bearings are written to a FontForge .pe script.

The build system runs kern-font.pe, which applies the output of the kerner program to the font. Something else that kern-font.pe does is to add a hardcoded additional bearing of 40 on the left and 80 on the right to all Japanese-script characters (kana and kanji); by trial and error, this seems to make the results look better. It seems to be simply a fact that Japanese characters need more space between them to look right than Latin characters do at the same type size. Something similar should probably be done to Korean-script characters, but that has not been determined yet.

That is how the horizontal spacing of the font is currently computed. It still isn't perfect, but some progress has been made.

さてさてなにが、できるかな？