作りましょう 0.2

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 0.2

Parametric Font Family
User Manual

Matthew Skala
mskala@ansuz.sooke.bc.ca

6 April 2011

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 capability, 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.

This is the second released version. Because it's 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 six ready-made faces, each of which can be drawn in monospace or proportional forms. In this version each style contains 1644 glyphs, up from 1306 in version 0.1 released February 19. That's 338 new glyphs added in 46 days; assuming that the font will be reasonably complete at 6000 glyphs, the projected completion date is near the end of the year 2012.

Current glyph coverage is the MES-1 subset of Unicode (corresponding to pretty complete coverage of the Latin script), the hiragana and katakana syllabaries, and a total of 198 kanji including the 80 taught at the Grade One level in the Japanese school system. So with these fonts you can write English pretty well; you can write all the most popular languages that use the Latin script (such as French, German, and Spanish, to name only a few; but not Greek or Russian); and you can write Japanese like a six-year-old. There are also some extra goodies, such as complete sets of I Ching hexagrams and Genjimon. More of that sort of thing will probably appear in future versions, as I'm throwing in characters that seem interesting whether they are particularly useful and necessary, or not.

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:

- Proportional spacing and kerning are new in this version, and heavily used in the package documentation because they look a lot nicer for English than the monospace versions of the fonts. Nonetheless, this feature remains quite experimental and unfinished. You should bear in mind that the horizontal spacing is all done automatically according to my own original theories on the subject embedded in the included "kerner" program; and some characters (especially punctuation marks like hyphen, and special symbols like ©) are currently just copies of the monospace versions and really should be redrawn to look right among the proportional alphabetic characters. Furthermore, proportional spacing of any kind is not traditional for Japanese typesetting, and a document written purely in Japanese may look better with the monospace fonts.

- 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.

- 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.

- Tsukurimashou Kaku is the main, baseline style, and only a limited amount of effort will be put into optimizing other styles.

- In the case of monospace: The "wide" forms of Latin glyphs are the baseline, even in cases of accented glyphs that Unicode doesn't include in wide form, and which appear in the fonts only in narrow form. The "wide" forms are the ones on which I will lavish the most attention; the "narrow" forms are mostly generated by an automated nonlinear scaling operation from "wide" designs. In the case of proportional space: "wide" and "narrow" glyphs are basically identical, though in the current version there are some differences in things like dollar sign ($ versus $).

- I do not currently plan to create an italic or slanted version; italics are not traditionally used in Japanese. However, I didn't plan to do proportional spacing either, and here it is, so who knows? For that matter, I've been playing with some blackletter ideas, too.

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

- The current XeTeX as of this writing, or possibly its fontspec package, contains a bug related to the combination of non-stretchable interword spaces with the LaTeX font size commands. Typesetting in monospace fonts in general is going to have problems until they fix that; but I've been in communication with the parties responsible and it appears that they will be fixing it before this typeface family is usable enough for it to be a big issue anyway.

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 file named LICENSE 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 over-riding 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 automatically 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. 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.

The remaining pages of this documentation file give some notes on the build system, which is much enhanced in this version, 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!"

よろしくおねがいします。

Matthew Skala
mskala@ansuz.sooke.bc.ca
April 6, 2011

『作りましょう』のつかいかた
Using Tsukurimashou

フォントファイル　　The Font Files

A complete build of the Tsukurimashou fonts will create a total of twelve font files in the otf subdirectory of the distribution. They all have names like "TsukurimashouKakuPS.otf," where "Tsukurimashou" is the meta-family name and the same for all of them, "Kaku" is the style name, and "PS" indicates that this is a proportionally spaced font. If there's no "PS," then it is a monospaced font. All these are Postscript-flavoured OpenType font files and should be compatible with most current word processing and typesetting software.

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　かく　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　アンビルてき　Tsukurimashou Anbiruteki ("Anvilicious"): extra-bold, rounded sans-serif.**
- 作りましょう　てんしのかみ　Tsukurimashou Tenshi no Kami ("Angel Hair"): very thin hairline display font.
- 作りましょう　ぼくっこ　Tsukurimashou Bokukko ("Tomboy"): felt marker style.
- 作りましょう　みんちょ　Tsukurimashou Mincho ("Ming Dynasty"): modern serif.

If you have just unpacked the distribution, there will only be Tsukurimashou Kaku and Tsukurimashou Mincho, each in its monospace and PS versions; instructions for building the others are in the subsection called "Building Tsukurimashou."

ほかのぶんすう　　Alternate Fractions (afrc)

As of version 0.2, fractions work properly only in the monospaced fonts.

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 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[

ともえのはながた　　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 "I TST." 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.

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," "Spıñal Tap," and "Mormoñ 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) → ㊿
- (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 ㋿, 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.

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 "Yumi 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: 

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

Combining I Ching dots only work properly in monospace fonts at present.

おおやけのユーログリフ　　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: €

せんもんてきマニュアル
Technical Documentation

Material in this section is not necessary for using the Tsukurimashou fonts, but may be of interest to those who wish to modify the fonts or understand more deeply how they work. As this is something of a hobby project, I'm throwing in a lot of material and technology of interest to me as a computer hobbyist.

『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 Slackware Linux. Anything branded as "Unix" should work. MacOS X might work. Windows with Cygwin might work.
- Perl.
- GNU Make.
- MetaType1, specifically the mtype13 distribution.
- A version of FontForge that actually works. This will probably require local patching, and possibly local debugging; see below.

Other things that might also be useful include:

- XeLaTeX (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 2010, then you probably have MetaType1 and XeLaTeX already. Note that it does have to be fairly up-to-date. The version that came with Slackware on my laptop had a bug that caused "❸" to appear as "{3}," where it should be a white "3" in a black circle. (It appears that the issue was with the braces being treated as "math" characters and substituted from Latin Modern instead of the specified Tsukurimashou font, so that as a result the OpenType substitution wasn't being triggered.)

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. I recommend compiling your own from recent development sources, with the "--enable-longdouble" option to configure, and also paying attention to this bug report I filed:
http://sourceforge.net/mailarchive/message.php?msg_id=26871721

I can't promise that applying that patch and turning on long doubles 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. But, hey, if you're lucky, maybe FontForge will have fixed all its own bugs by the time you read this and you'll get along all right with the distributed version!

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; 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, or your system's dict or share/dict directories, possibly under /usr or /usr/local or your configured installation prefix; 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 (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) to temporarily disable the silent build rules and see what's going on. The third-party font-building programs unconfigurably produce large quantities of verbose messages, some sent inexcusibly to /dev/tty instead of standard output or error, and the build system will go to heroic lengths with Expect (if available) to filter those out by default.

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 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

[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.

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 completely untested and probably will not work in this version. I hope to make it work in some future version. For the moment it is safer to build right in an unzipped copy of the distribution. The "dist-check" target will almost certainly fail because of the lack of separate build directory support.

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.

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.

At the moment, the configure script looks for SWI-Prolog, ECL^{i}PS^{e}-CLP, and GNU Prolog, these three; but the greatest of these, and the only one the Makefiles can actually use at the moment, is SWI. Support for the other two will probably be in the next release. ECL^{i}PS^{e}-CLP has the minor 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. I plan to make some other checks, and if it's not possible to identify ECL^{i}PS^{e}-CLP with confidence, it'll just give the user a warning and let them tell configure if they want to risk it. Some of the code for such a warning is already in place and you may see it if you have an executable called "eclipse" on your system.

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 sub-goals 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.

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 5 font units (half a pixel) 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 so you don't end up setting the next character actually earlier on the line.

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 k-means classification. That means they are initially put into 150 classes (by a simple round-robin: first glyph in class 1, second in class 2, on to 150th in class 150, 151st in class 1 again, and so on), and 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. 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 150 classes of left contours and 150 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 22500 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 150 seems to be about as big as I can make it without FontForge starting to complain about tables being too large; it is possible that paying more attention to "subtable" division could stretch this number larger, but that's not a gimmie because this table (unlike typical human-designed tables) is dense, with a kern between each class and a large fraction of all other classes. Because we are in theory kerning every character to every other character, there's no block structure such as would come from only kerning pairs we think are likely to occur in text. As I understand it, the point of subtables is to exploit block structure.

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), 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 effect of the exponent 3 in the calculation is to give extra weight to points that are close together. 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 rounded to the nearest multiple of 12 font units and written to a feature file. The rounding is to eliminate the many entries that are very close to zero, and possibly enable better file compression by reducing the number of distinct values of entries; I still plan to do some experiments to see whether just leaving out small entries and not rounding others, would be better. 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.

That is how the horizontal spacing of the font is currently computed. It still isn't perfect; in particular, I'd like to add some overrides for things like combining characters, whose negative bearings are completely screwed up by this process. The fractions feature, which depends on all the fraction characters having the same width (it's basically implemented as a bunch of combining characters) also needs work. And I'm still not sure what to do about characters for which white space can be thought of as part of the character (for instance, ideographic fullstop). But this is a start.

さてさてなにが、できるかな？