UNOFFICIAL

The OpenType Cookbook

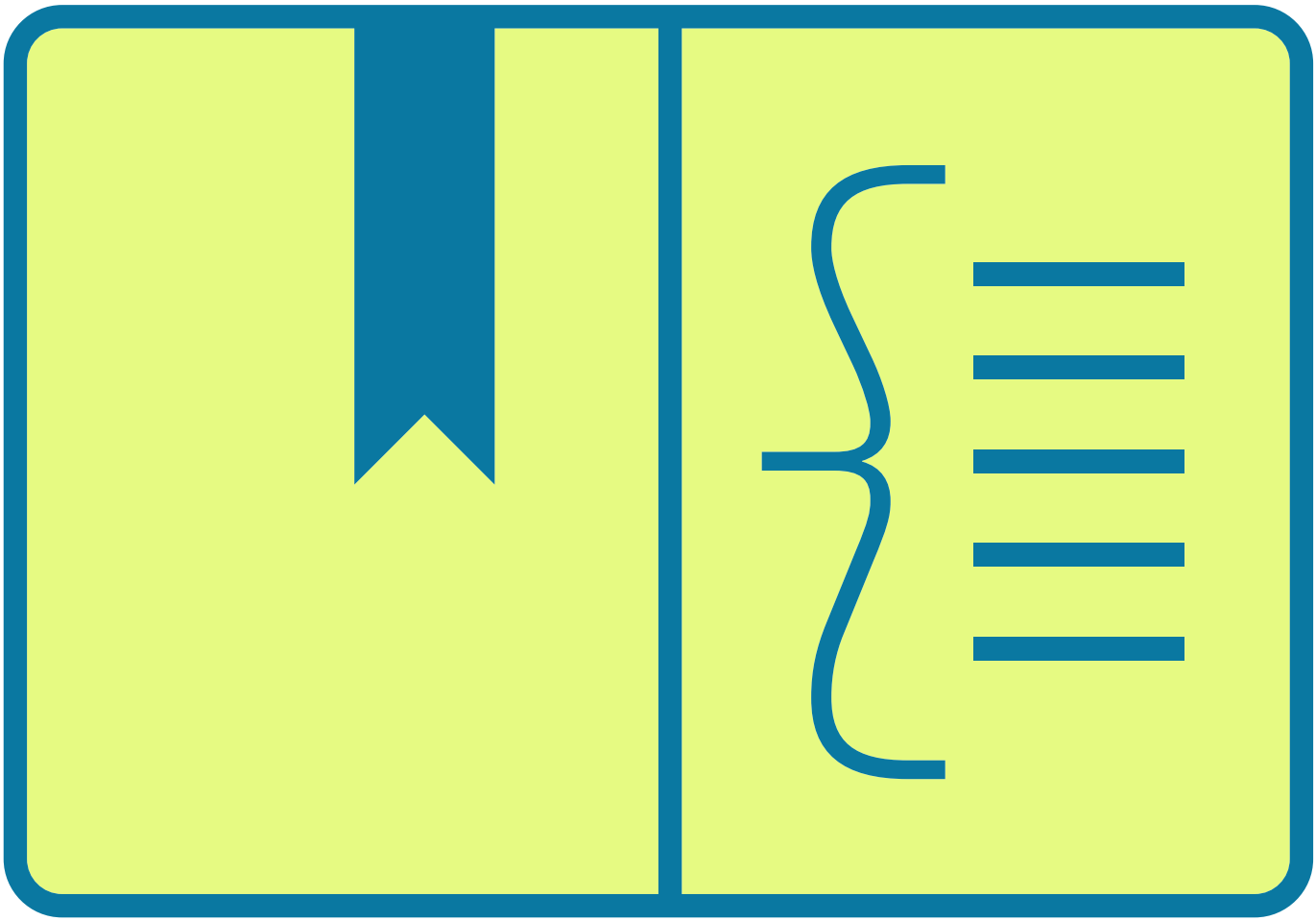


Table of Contents

Introduction

OpenType features allow fonts to behave smartly. This behavior can do simple things (e.g. change letters to small caps) or they can do complex things (e.g. insert swashes, alternates, and ligatures to make text set in a script font feel handmade). This cookbook aims to be a designer friendly introduction to understanding and developing these features. The goal is not to teach you how to write a small caps feature or a complex script feature. Rather, the goal is to teach you the logic and techniques for developing features. Once you understand those, you'll be able to create OpenType features that fit your design as perfectly as possible.

This cookbook is written with the assumption that you have a basic working knowledge of the structure of a font. You need to know the differences between characters and glyphs, understand the coordinate system in glyphs and so on. If you aren't familiar with that, go study the basics and then come back. This will all make a lot more sense that way.

Your Author

I'm Tal Leming and I started developing OpenType features around 2002 or 2003. Since then I have developed the OpenType features for a number of fonts that do weird things. I only write OpenType features for my own fonts now, so I am semi-retired from day-to-day OpenType feature programming. However I teach type design at MICA and a number of my students have expressed interest in developing their own OpenType features and there wasn't a good from the ground up introduction to this and...here we are. (Hi to my students!)

Unofficial

This cookbook is not written by, sponsored by, supported, endorsed or whatever by the inventors of the OpenType features data storage formats and execution protocols. (That would be Adobe and Microsoft). This cookbook is unofficial unofficial unofficial. Did you read the part about who I am and why wrote this?
To repeat, this is unofficial.

Getting Help

If you have trouble understanding anything, need help with your features or anything like that, please post your questions on the UAFDKOML (Unofficial Adobe Font Developent Kit For OpenType Mailing List) or start a discussion on Type Drawers. I'm unfortunately not able to help with questions directly due to time constraints.

Feedback & Contributions

This source for the text of this cookbook is on GitHub. If you spot errors, have suggestions about making something more clear and so on, please open an issue or submit a pull request over there. If you would rather get in touch with me directly, feel free to do that here.

Donations

If you think this cookbook has been useful in your day to day work and you want to support the development of it, feel free to drop a tip or donation into the hat over on Pledgie. Or you can contact me directly if you would prefer to do it that way.

Students

If you are a student, please don't donate. Save your money for food, tuition, supplies, hardware, software, fonts, picnics, concerts, student loans, etc. Come back when you are rich and famous from all the world changing OpenType features that you have written. I mean, if you really, really, really want to donate, you can if you insist. But, keep your budget in mind, please.

1 | Foundation Concepts

Before we start writing any code, let's first look at what the code will actually do. If you are reading this cookbook just to find out how to do something simple like write a small caps feature, you may skip this section and jump ahead to the code snippets. But, if you want to develop complex, nuanced, amazing features, you really should read this section. Understanding the underlying mechanics of how features work will allow you to carry your vision all the way from how the glyphs the glyphs look to how the behave.

Ready? Alright, let's get into some heavy stuff.

Structures

In OpenType we can define behaviors that we want to happen upon request from users. For example, the user may decide that text should be displayed with small caps. You, the type designer, can define which glyphs should be changed when this request is made by the user. These behaviors are defined in *features*. Features can do two things: they can *substitute glyphs and they can adjust the positions of glyphs.*

The actual behavior within the features are defined with rules. Following the small caps example above, you can define a rule that states that the `a` glyph should be replaced with `A.sc` .

Within a feature, it is often necessary to group a set of rules together. This group of rules is called a *lookup*.

Visually, you can think of features, lookups and rules like this:

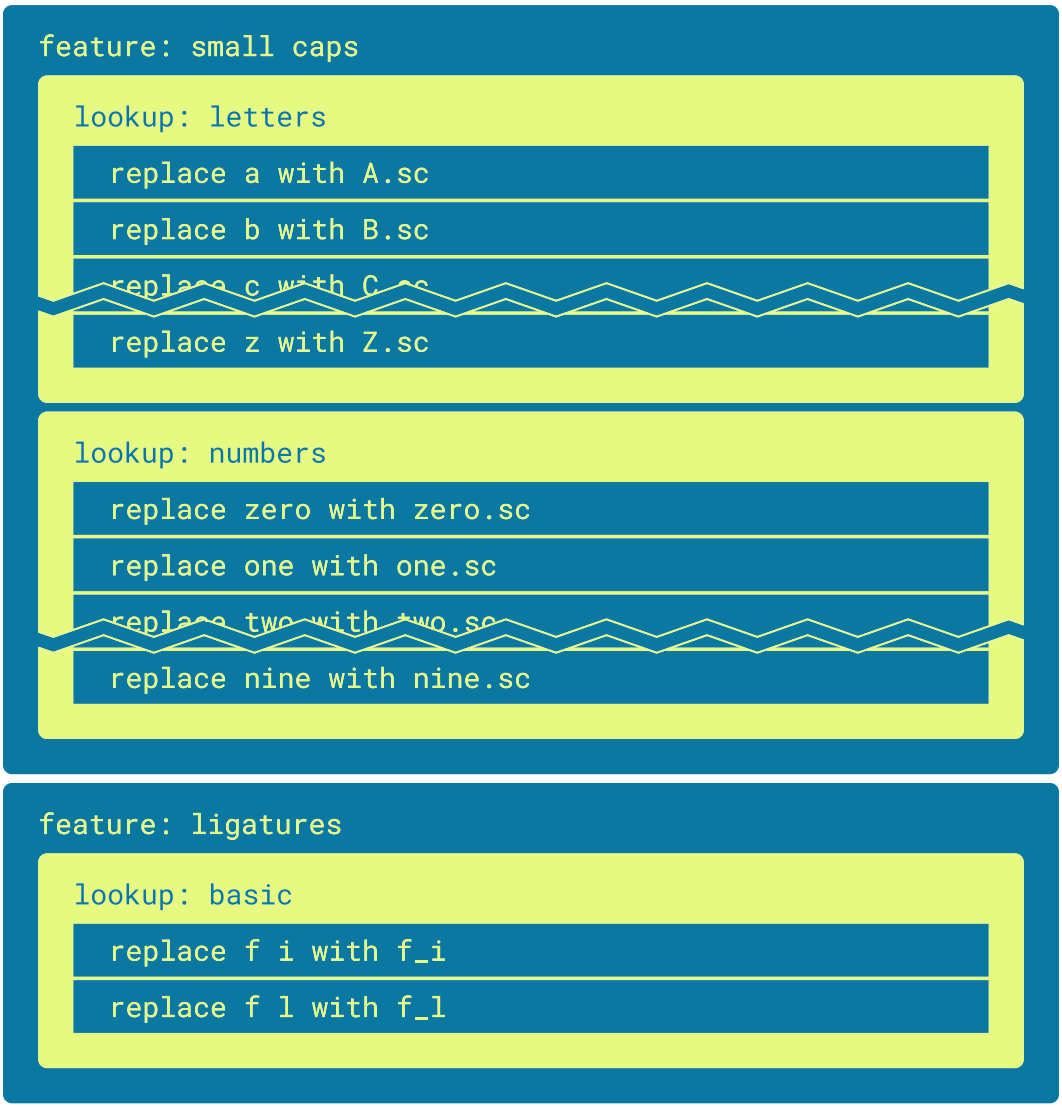
(Note: In these illustrations if you see a jagged line cutting something off, it means "There is a bunch of the same kind of stuff so we'll cut it off to avoid too much repetition.")

Processing

When text is processed, the features that the user wants applied are gathered into two groups: substitution features and positioning features. The substitution features are processed first and then the positioning features are processed. The order in which you have defined the features, lookups and rules is the order in which they will be applied to the text. This order is *very* important.

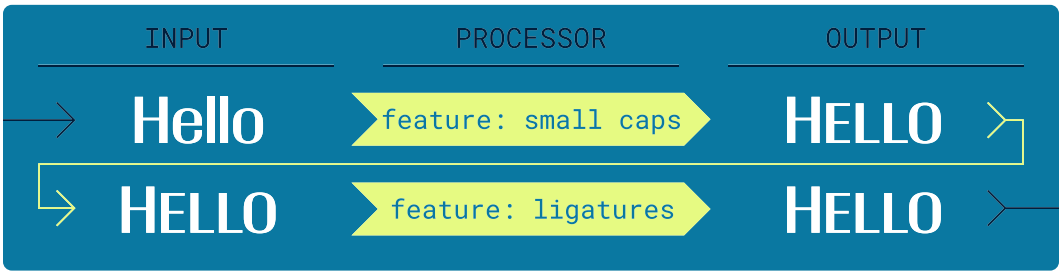
Features process sequences of glyphs. These glyph runs may represent a complete line of text or a sub-section of a line of text.

For example, let's assume that you have the following features, lookups and rules:

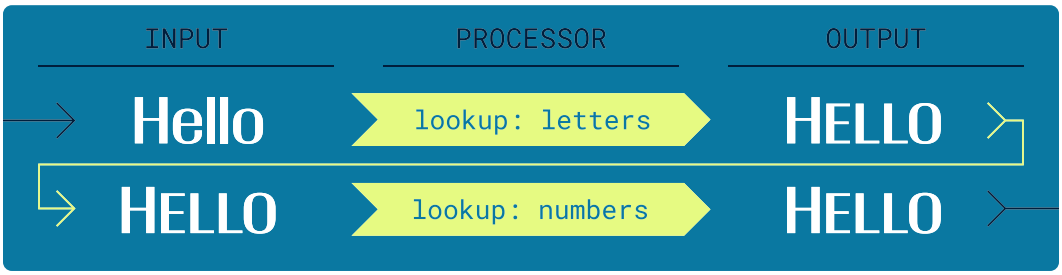


Let's also assume that the user wants to apply small caps and ligatures to a glyph run that displays `Hello`.

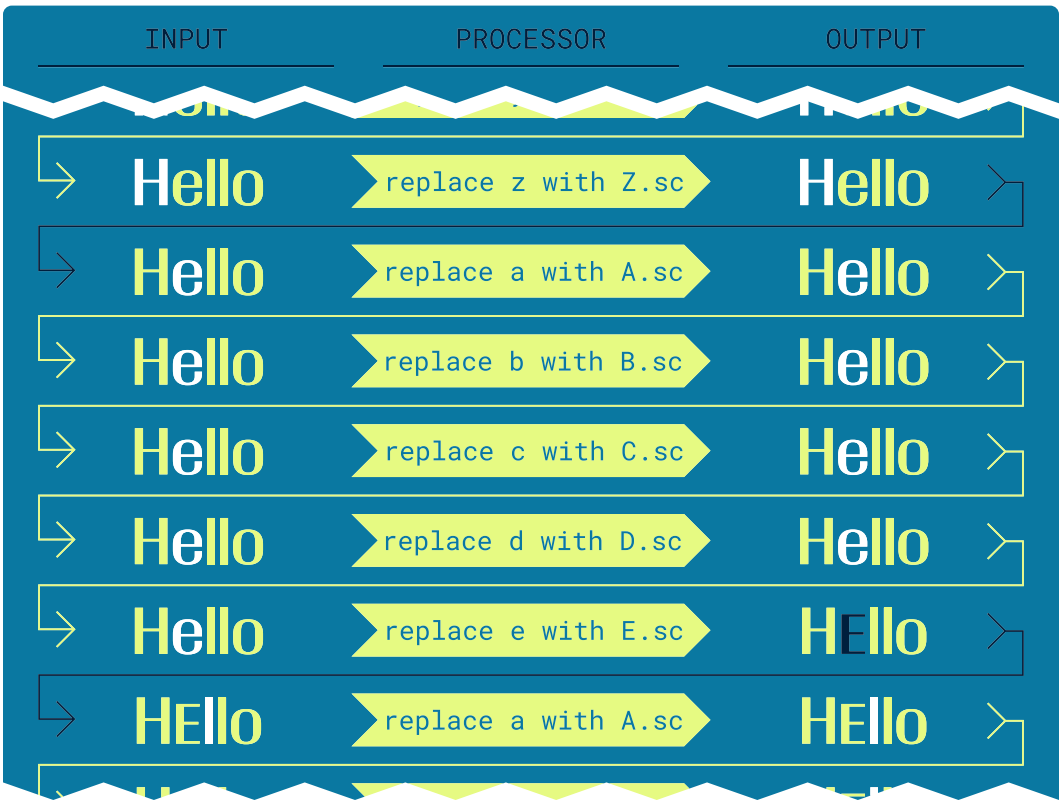
A glyph run is processed one feature at a time. So, here is what `Hello` will look like as it enters and exists each feature:



Within each feature, the glyph run is processed one lookup at a time. Here is what our example looks like as it moves through the small caps feature:



Within each lookup, things are a little different. The glyph run is passed one glyph at a time from beginning to end over each rule within the lookup. If a rule replaces the current glyph, the following rules are skipped for the current glyph. The next glyph is then current through the lookup. That's complex, so let's look at it with our example:



The process is the same for positioning features, except that instead of rule evaluation stopping when a glyph is replaced, the evaluation is stopped when a glyph's position is changed.

That's how processing works and it is the most complex part of OpenType features that you will need to understand. Got it? Great!

2 | Syntax Introduction

We will be writing our features in the Adobe OpenType Feature File Syntax (commonly referred to as ".fea"). This is a simple, text format that is easily editable in text and font editors. There are other syntaxes and tools for developing features, but .fea is the most widely supported and the most easily accessible. We'll be going through the important parts of .fea in detail, but for now we need to establish some basics.

Comments

It's useful to be able to write comments about your code. To do this, add a `#` and everything from the `#` to the end of the line of text will be marked as a comment.

```
# This is a comment.
```

Comments are ignored when your font is compiled, so you can write anything you want in your comments.

White Space

In some syntaxes the amount of white space is important. This is not the case in .fea. You can use spaces, tabs and line breaks as much and in any combination as you like.

Special Characters

Some characters have special meanings in .fea.

`;` A semicolon indicates the closing of something–a feature, lookup, rule, etc. These are important.

`{ }` Braces enclose the contents of a feature or lookup.

`[ ]` Brackets enclose the contents of a class.

Features

Features are identified with a four character tag. These are either **registered tags** or private tags. Unless you have a very good reason to create a private tag, you should always use the registered tags. Applications that support OpenType features use these tags to identify which features are supported in your font. For example, if you have a feature with the `smcp` tag, applications will know that your font supports small caps.

Features are defined with the feature keyword, the appropriate tag, a pair of braces, the tag again, and a semicolon.

```
feature smcp {
      # lookups and rules go here
} smcp;
```

Lookups

Lookups are defined in a similar way to features. They have a name, but the name is not restricted to four characters or to a tag database. You can make up your own name, as long as it follows the general naming rules.

```
lookup Letters {
      # rules go here
} Letters;
```

Classes

You'll often run into situations where you want use a group of glyphs in a rule. These groups are called classes and they are defined with a list of glyph names or class names inside of brackets.

```
[A E I O U Y]
```

Classes can have a name assigned to them so that they can be used more than once. Class names follow the general naming rules and they are always preceded with an `@` . To create a named class you set the name, then an `=` , then the class definition and end it with a semicolon.

```
@vowels = [A E I O U Y];
```

After a class has been defined, it can be referenced by name.

```
@vowels
```

General Naming Rules

A name for a glyph, class or lookup must adhere to the following constraints:

- No more than 31 characters in length.
- Only use characters in A-Z a-z 0-9 . _
- Must not start with a number or a period.

You should avoid naming anything (including glyphs) with the same name as a **reserved keyword**. If you do need to name a glyph with one of these names, precede a reference to the glyph with a `\` But, really, try to avoid needing to do this.

3 | Rules

Now that we have introduced some terminology, covered the way text is processed and established the general syntax rules, we can get into the fun part: actually doing stuff.

Substitutions

Substitutions are the most visually transformative thing that features can do to text. And, they are easy to understand. There are two main parts to a substitution:

1. Target – This is what will be replaced.
2. Replacement – This is what will be inserted in place of the target.

The syntax for a substitution is:

```
substitute target by replacement;
```

We can abbreviate `substitute` with `sub` to cut down on how much stuff we have to type, so let's do that:

```
sub target by replacement;
```

Targets and replacements can often be classes. These classes can be referenced by name or they can be defined as an unnamed class inside of a rule.

Replace One with One

To replace one thing with another, you do this:

```
sub target by replacement;
```

(In the .fea documentation, this is known as GSUB Lookup Type 1: Single Substitution.)

For example, to transform `a` to `A.sc` , you would do this:

```
sub a by A.sc;
```

If you want to replace several things with corresponding things, you can use classes as both the target and the replacement. However, in this case the number of things in the two classes needs to be the same, unlike above.

```
sub [a b c] by [A.sc B.sc C.sc];
```

It's usually more readable to define the classes earlier in your code and then reference them by name.

```
sub @lowercase by @smallcaps;
```

The order of the glyphs in your classes in this situation is critical. In the example above, the classes will correspond with each other like this:

```
a → A.sc
b → B.sc
c → C.sc
```

If you order the target and replacement classes incorrectly, things will go wrong. For example, if you have this as your rule:

```
sub [a b c] by [B.sc C.sc A.sc];
```

The classes will correspond like this:

```
a → B.sc
b → C.sc
c → A.sc
```

This is obviously undesired behavior, so keep your classes ordered properly.

Replace Many With One

To replace a sequence of things with one thing, you do this:

```
sub target sequence by replacement;
```

(In the .fea documentation, this is known as GSUB Lookup Type 4: Ligature Substitution.)

For example, for an `fi` ligature, you would do this:

```
sub f i by f_i;
```

You can also use classes as part of the target sequence:

```
sub @f @i by f_i;
```

Or:

```
sub @f i by f_i;
```

Or:

```
sub f @i by f_i;
```

Replace One With Many

To replace a single thing with a sequence of things, you do this:

```
sub target by replacement sequence;
```

(In the .fea documentation, this is known as GSUB Lookup Type 2: Multiple Substitution.)

For example, to convert an `fi` ligature back into `f` and `i`, you would do this:

```
sub f_i by f i;
```

Classes can't be used as the target or the replacement in this rule type.

Replace One From Many

To give the user a choice of alternates, you do this:

```
sub target from replacement;
```

(In the .fea documentation, this is known as GSUB Lookup Type 3: Alternate Substitution.)

The replacement must be a glyph class, and (unlike above) does not happen automatically, but usually requires active user interaction (e.g. picking glyphs from a selection of alternates).

For example, to give the user several options to replace `a` with, you would do this:

```
sub a from [a.alt1 a.alt2 a.alt3];
```

Note that the keyword in the middle of the rule is `from` instead of `by`.

Positioning

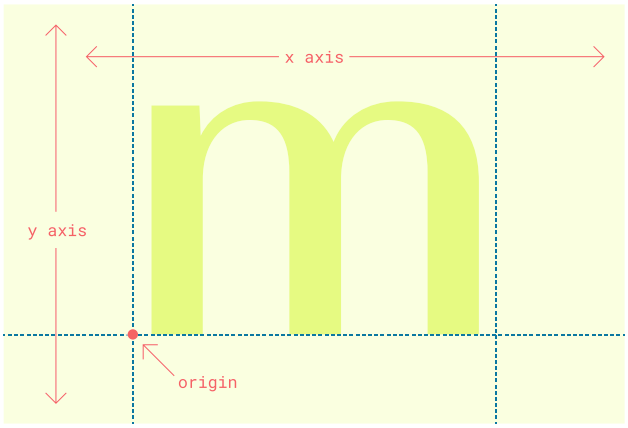
Positioning glyphs may not be as visually interesting as what can be achieved with substitution, but the positioning support in OpenType is incredibly powerful and important. The positioning rules can be broken into two separate categories:

1. Simple Rules – These adjust either the space around one glyph or the space between two glyphs.
2. Mind-Blowingly Complex and Astonishingly Powerful Rules – These do things like properly shift combining marks to align precisely with the base forms in Arabic and Devanagari so that things look incredibly spontaneous and beautiful.

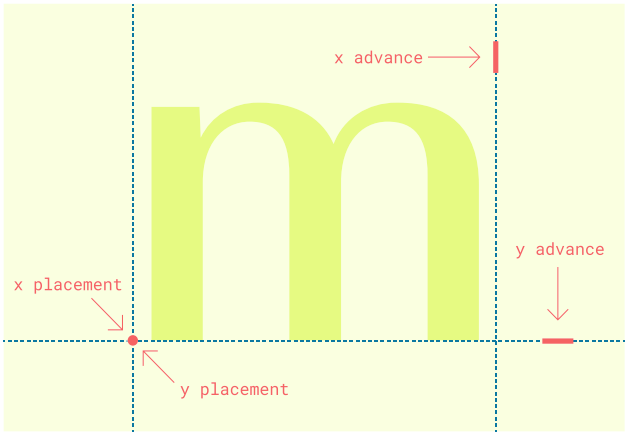
We're going to cover the simple rules in this cookbook. The complex rules are amazing, but too advanced for now.

Position and Advance

Before we go much further we need to talk about coordinate systems and value records. As you know, the coordinate system in fonts is based on X and Y axes. The X axis moves from left to right with numbers increasing as you move to the right. The Y axis moves from bottom to top with numbers increasing as you move up. The origin for these axis is the intersection of the 0 X coordinate, otherwise known as the baseline, and the 0 Y coordinate.



In the positioning rules, we can adjust the *placement* and *advance* of glyphs. The placement is the spot at which the origin of the glyph will be aligned. The advance is the width and the height of the glyph from the origin. In horizontal typesetting, the height will be zero and the width will be the width of the glyph. The placement and advance can each be broken down into X and Y values. Thus, there is an x placement, a y placement, an x advance and a y advance.



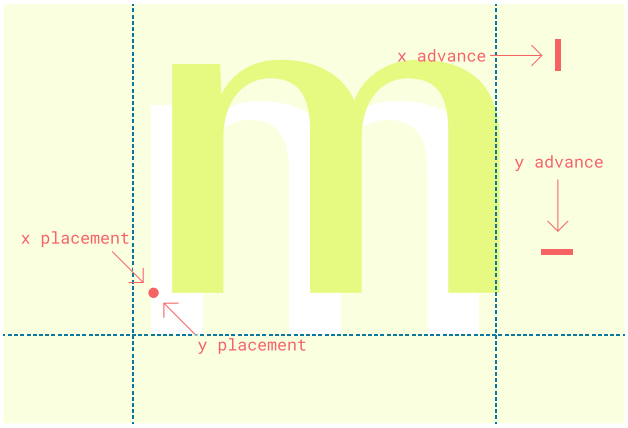
The units that these values represent are the same units in which you have drawn your glyph. Together, these four values form a *value record*. In the .fea syntax, we express these value records like this:

```
<xPlacement yPlacement xAdvance yAdvance>
```

For example:

```
<10 20 30 40>
```

In this case, the value record is adjusting the x placement to the right by 10 units, the y placement up by 20 units, the x advance by 30 units and the y advance by 40 units.



The syntax for a positioning rule is:

```
position target valueRecord;
```

We can abbreviate `position` with `pos` to cut down on how much stuff we have to type, so let's do that:

```
pos target valueRecord;
```

Targets can be classes. These classes can be referenced by name or they can be defined as an unnamed class inside of a rule.

Cumulative Effect

When the positioning features are started, each glyph in the glyph run has a value record of `<0 0 0 0>` . As the processing happens and rules are matched, these value records are modified cumulatively. So, if one feature adjusts a glyph's x placement by 10 units and then another feature adjusts the glyph's x placement by 30 units, the glyph's x placement would be 40 units.

Adjust the Position of One Glyph

To adjust the space around a single target, you do this:

```
pos target valueRecord;
```

(In the .fea documentation, this is known as GPOS Lookup Type 1: Single Adjustment Positioning.)

For example, to put some space to the left and right of the `A` , you would do this:

```
pos A <10 0 20 0>;
```

Adjust the Space Between Two Glyphs

To adjust the space between two targets, you do this:

```
pos target1 target2 valueRecord;
```

(In the .fea documentation, this is known as GPOS Lookup Type 2: Pair Adjustment Positioning.)

In this case, you can shorten the value record to be only the x advance adjustment. Or, you can use the full value record if you prefer that.

This rule is used almost exclusively for kerning. In fact, this is so common that you shouldn't have to write any of these rules yourself. Your font and/or kerning editor should do this for you.

You can use a class as target 1, target 2 or both:

```
pos @A T -50;
```

Or:

```
pos A @T -50;
```

Or:

```
pos @A @T -50;
```

But, seriously, let your editor write these rules for you.

Substitutions and Positioning Based on Context

The substitution and positioning rules that we have discussed so far are quite useful, but the real power is in triggering these rules only when certain conditions are met. These are known as contextual rules.

Contextual rules allow us to specify a sequence before the target, a sequence after the target or both in a substitution or positioning rule. For example: replace `r` with `r.alt` if the `r` is preceded by wo and followed by `ds` . There are two new parts of this rule type in addition to the parts we defined in the substitution and positioning sections.

1. Backtrack – This is the sequence of things that occur before the target in the rule. This sequence can be composed of glyphs, classes or a mix of both.
2. Lookahead – This is the sequence of glyphs that occur after the target in the rule. Like the backtrack, this sequence can be composed of glyphs, classes or a mix of both.

The backtrack and lookahead are both optional. Either, or neither, can appear. If a sequence is present, it can contain one or more things.

In addition to the backtrack and lookahead, a new character is needed in these rules: `'` . This character is used to mark the target of the rule.

Here is the words example from above in the correct syntax:

```
sub w o r' d s by r.alt;
```

Most of the substitution and positioning rule types can be defined with a context.

- replace one with one: `sub a b' c by b.alt;`
- replace many with one: `sub a b' c' d by b_c;`
- adjust position of one glyph: `pos A B' C <10 0 20 0>;`
- adjust positioning of the space between two glyphs: `pos A B' C' -50 D;`

Please note that just because you *can* apply this to a rule type doesn't mean that it always makes sense; or that you should.

Exceptions

What if we have a short context that you want to match, but a longer context that contains the short context? For example, say we want to change the r in `words` but not in `words!` . To do that we can specify an *exception* to the contextual rule. For example:

```
ignore sub w o r' d s exclam;
sub w o r' d s by r.alt;
```

The `ignore` keyword followed by a backtrack (optional), target and lookahead (optional) creates the exception.

Common Gotcha

If you use a contextual rule or exception within a lookup, all of the rules within that lookup must also use the `'` on the target of the rule. For example:

```
sub a b' c by b.alt;
sub d' t d.alt;
```

Advanced Techniques

By now we have established the rules needed to make most features that you'll want to add to your fonts–small caps, ligatures, tabular figures, etc. But, when you want to do some more complex things, you'll need a few more things.

Language Systems

One of OpenType's best attributes is the way that it handles languages and scripts. We can define rules that only apply when the user has indicated that they are writing a particular language or using a particular script. To do this, we state the script and the language that the rules apply to. After these have been made, all subsequent rules in the feature belong to this language and script unless you declare another language and script.

Let's look at an example. Let's say that we have a special `IJ` that should only be used when Dutch is the declared language. We need to say: when the script is latin and the language is Dutch, replace `IJ` with `IJ.alt` . Here is how we do that:

```
script latn;
language NLD;
sub IJ by IJ.dutch;
```

The script tags are defined **here** and language tags are defined **here**.

If you add language or script specific rules you also need to register that the features include a particular language and script combination, known as a language system, before any of your feature definitions. This is the syntax:

```
languagesystem script language;
```

And in our example, we would do this:

```
languagesystem latn NLD;
```

Before you define any specific language system, you should always declare this:

```
languagesystem DFLT dflt;
```

This will register all rules for a fallback system in case an OpenType layout engine gets confused about which language or script your features apply to. Additionally, before you register a script with a specific language, you should register it with the default language for the same reason. So, the complete set of language system statements would look like this:

```
languagesystem DFLT dflt;
languagesystem latn dflt;
languagesystem latn NLD;
```

Lookups

We studied lookups when we went through the feature processing model, but they deserve some extra emphasis. Multiple lookups are allowed in a single feature and the fact that they process an entire glyph run before moving on to the next lookup makes them incredibly useful. We can use this technique to produce some very complex behavior. For example, in a swash feature it is often best to insert swashes first at the beginning and then at the end of words in separate passes. We can do this easily with lookups.

You can also reuse lookups if they are declared outside of a feature. To do this, define your lookup like this:

```
lookup Example {
    # rules go here
} Example;
```

Then, inside of your features you can have this lookup called by referencing its name:

```
lookup Example;
```

This is useful if you want to share some rules across multiple features.

```
lookup Inferiors {
    sub @inferiorOff by @inferiorOn;
} Inferiors;

feature subs {
    lookup Inferiors;
} subs;

feature sinf {
    lookup Inferiors;
} sinf;
```

4 | Putting it Together

You now have the building blocks for writing everything from simple to complex features and you understand how it is actually going to work. Right? Great! Now, let's start looking at the bigger picture.

All of your features will be in a text file somewhere, either stored in your font source file or in an external file. I like to structure the code in my files like this:

1. languagesystem declarations
2. global classes
3. all substitution features
4. all positioning features

The language system declarations must come first. After that, the order is up to you, but I like to add any classes that I will use frequently, then all of the substitution features and then all of the positioning features.

Building a Feature

Here is a basic example combined into the proper form:

```
languagesystem DFLT dflt;
languagesystem latn dflt;

@lowercase = [a    b    c];
@uppercase = [A    B    C];
@smallcaps = [A.sc B.sc C.sc];

@figures         = [zero    one    two];
@figuresSmallcap = [zero.sc one.sc two.sc];

@punctuation         = [exclam];
@punctuationSmallcap = [exclam.sc];

feature smcp {
      sub @lowercase by @smallcaps;
} smcp;

feature c2sc {
      sub @uppercase by @smallcaps;
      sub @figures by @figuresSmallcap;
      sub @punctuation by @punctuationSmallcap;
} c2sc;
```

Note that there are no lookups declared. Why? The first lookup is implied to be anything before you define a lookup. Consider this example:

```
feature c2sc {
      sub @uppercase by @smallcaps;

      lookup Lowercase {
            sub @lowercase by @smallcaps;
      } Lowercase;
} c2sc;
```

The first rule, `sub @uppercase by @smallcaps;` , is implicitly in a lookup. The result is the same as this:

```
feature c2sc {
      lookup Uppercase {
            sub @uppercase by @smallcaps;
      } Uppercase;

      lookup Lowercase {
             sub @lowercase by @smallcaps;
      } Lowercase;
} c2sc;
```

That's pretty much how you combine things. There will be more examples that demonstrate advanced techniques that you can study later.

Feature Order

The order in which you list your features is very important. This is the order in which they will be processed. If you do something like put your ligatures before any alternates, your alternate code will have to take into account potential ligatures. It may even possibly have to break the ligatures down. That's possible, but it results in overly complex code that is hard to read, edit and debug. I generally order my features like this:

1. script language specific forms (locl)
2. fractions (frac, numr, dnom)
3. superscript and subscript (sups, subs)
4. figures (lnum, onum, pnum, tnum)
5. ordinals (ordn)
6. small caps (smcp, c2sc)
7. all caps (case)
8. various alternates (cswh, titl, salt, ss01, ss02, ss...)
9. ligatures (liga, dlig)
10. manual alternate access (aalt)
11. capital spacing (cpsp)

There are, of course, exceptions. The point is that you should think through this ordering so that you don't make things harder on yourself than you have to.

Including an External File

If you are working on a family that shares features across multiple styles, it's cumbersome to store the features in each font source file. To get around this, you can store the features in an external file and reference them from the features in your font source file. To do this you use the `include` keyword:

```
include(features/family.fea);
```

The text inside of the parenthesis must be the path to your external features file relative to the font source file. In the example above, the file family.fea is located in a folder called features right next to my font source file.

You can even include multiple files in a single font source file. For example:

```
include(features/family.fea);
include(features/bold-kern.fea);
```

Even in other folders, below the current one:

```
include(../tables.fea);
```

5 | Common Techniques

Up until now, almost everything that has been discussed has been the official way to do things or established common practice. The next section is far more subjective as there are numerous ways to write the code to create certain behaviors. What follows includes my own personal opinion.

I have prepared a **demo font** that includes all of the features and techniques defined below. You can open it up in your favorite font editor and play around with the features if you want to see how they work. The feature tags in the code below don't always match the tags in the font. I had to do it this way since there can't be more than one version of each feature.

The features in the demo font are by no means complete or an indication of what is necessary to include in a particular feature. You should develop your own interpretation of what should be included in each feature based on the design of your font.

Glyph Run and Word Boundary Detection

Often we want to make substitutions based on the bounds of words and glyph runs. For example, to insert a swash only at the beginning or at the end of a word. There are three features in the specification for dealing with these situations:

- `init` – Performs substitutions only at the beginning of a word.
- `fina` – Performs substitutions only at the end of a word.
- `medi` – Performs substitutions only on the glyphs between the first and last in a word.

Unfortunately the specification is a bit vague about how these are supposed to be implemented. The Unicode specification details a word boundary detection algorithm and conceivably that's what would be used by the layout engines that are processing your font. That specification is quite thorough, but it thinks about word boundaries in a different way than type designers do (or at least the type designer writing this does). For example, where are the word boundaries in this text?

- Hello "World!"

They are at the `o` , the `W` and the `d` . If we use this for swashes our `W` and `d` are likely to clash with the marks `"` and `!` around them. We often think of word boundaries as an empty space around words. If we want to use `init` and `fina` , we'll need to build in exceptions. You can certainly do that, but I generally do it all myself with some special classes:

- `@all` – This class contains all glyphs.
- `@filled` – This class contains all glyphs that contain positive space.
- `@empty` – This glyph contains all glyphs that contain only negative space.

With these, we can build lookups that handle boundary detection reasonable well enough for things like swashes.

```
lookup GlyphRunInitial {
      ignore sub @all @initialsOff';
      sub @initialsOff' by @initialsOn;
} GlyphRunInitial;

lookup GlyphRunFinal {
      ignore sub @finalsOff' @all;
      sub @finalsOff' by @finalsOn;
} GlyphRunFinal;

lookup WordInitial {
      ignore sub @filled @initialsOff';
      sub @initialsOff' by @initialsOn;
} WordInitial;

lookup WordMedial {
       sub @filled @medialOff' @filled by @medialOn;
} WordMedial;

lookup WordFinal {
      ignore sub @finalsOff' @filled;
      sub @finalsOff' by @finalsOn;
} WordFinal;
```

To be clear, you should not use this for shaping Arabic or anything like that. This is strictly for aesthetic substitutions like swashes.

Script and Language Specific Forms

The `locl` feature is specifically designed to implement global script and language specific changes that need to happen before any other features are processed. You can certainly put stylistic specific changes in other features, but the big important ones should be in `locl`.

```
feature locl {

	script latn;

		language NLD exclude_dflt;
			lookup DutchIJ {
				sub IJ by IJ.dutch;
			} DutchIJ;

} locl;
```

Fractions

I have used two different algorithms for implementing on-the-fly fractions. The first is fairly straightforward. The second is more complex but I think it is easier for users.

Method 1: Individual

This method has been around for as long as I have been working on OpenType features. Adobe probably developed it in the very early days of the .fea language. It is probably still the most common implementation.

```
feature frac {
	@slash = [slash fraction];

	lookup FractionNumerators {
		sub @figures by @figuresNumerator;
	} FractionNumerators;

	sub [@slash @figuresDenominator] @figuresNumerator' by @figuresDenominator;
	sub slash by fraction;
} frac;
```

Method 2: Contextual

Around 2006 **Kent Lew** asked me if I had any ideas for a better fraction implementation. Specifically, he was referring to the fact that with the existing implementation users had to manually select only the text that should be converted to fractions and apply the feature. If the feature was applied to more than just that text all numbers not in a fraction would be converted to numerators. This was a big problem in things like cookbooks where there could be thousands of little bits of text that had to be converted to fractions.

I developed a new method that is built on the common form of writing fractions as an integer, a space, a numerator, a slash and a denominator. For example: `2 1/2`. The code considers 1-10 numbers followed by a slash followed by 1 or more numbers to be a fraction. The slash is converted to a fraction bar, the numbers before the slash are converted to numerators and the numbers after the slash are converted to denominators. If the new fraction is preceded by a number followed by a space, the space is converted to a thin space to pull the fraction closer to the integer. After I published the first version of this code, **Karsten Luecke** pointed out some problems with dates, German tax numbers and things like that. I published a new version that handles these properly and this version is on the following page.

With this users can *globally* activate fractions. The only drawback that I have found with this is that it doesn't allow numerators to be longer than 10 numbers long. In the unlikely event that a user runs into this problem, they can select the unconverted numerators and activate the numerator feature.

```
feature frac {

    lookup FractionBar {
        ignore sub slash @figures @figures @figures @figures @figures @figures @figures @figures @figures
         slash';
        ignore sub slash' @figures @figures @figures @figures @figures @figures @figures @figures @figures
         slash;
        ignore sub slash @figures @figures @figures @figures @figures @figures @figures @figures slash';
        ignore sub slash' @figures @figures @figures @figures @figures @figures @figures @figures slash;
        ignore sub slash @figures @figures @figures @figures @figures @figures @figures slash';
        ignore sub slash' @figures @figures @figures @figures @figures @figures @figures slash;
        ignore sub slash @figures @figures @figures @figures @figures @figures slash';
        ignore sub slash' @figures @figures @figures @figures @figures @figures slash;
        ignore sub slash @figures @figures @figures @figures @figures slash';
        ignore sub slash' @figures @figures @figures @figures @figures slash;
        ignore sub slash @figures @figures @figures @figures slash';
        ignore sub slash' @figures @figures @figures @figures slash;
        ignore sub slash @figures @figures @figures slash';
        ignore sub slash' @figures @figures @figures slash;
        ignore sub slash @figures @figures slash';
        ignore sub slash' @figures @figures slash;
        ignore sub slash @figures slash';
        ignore sub slash' @figures slash;
        ignore sub slash slash';
        ignore sub slash' slash;
        sub @figures slash' @figures by fraction;
    } FractionBar;

    lookup Numerator1 {
        sub @figures' fraction by @figuresNumerator;
    } Numerator1;

    lookup Numerator2 {
        sub @figures' @figuresNumerator fraction by @figuresNumerator;
    } Numerator2;

    lookup Numerator3 {
        sub @figures' @figuresNumerator @figuresNumerator fraction by @figuresNumerator;
    } Numerator3;

    lookup Numerator4 {
        sub @figures' @figuresNumerator @figuresNumerator @figuresNumerator fraction by @figuresNumerator;
    } Numerator4;

    lookup Numerator5 {
        sub @figures' @figuresNumerator @figuresNumerator @figuresNumerator @figuresNumerator fraction by
         @figuresNumerator;
    } Numerator5;

    lookup Numerator6 {
        sub @figures' @figuresNumerator @figuresNumerator @figuresNumerator @figuresNumerator @figuresNumerator
         fraction by @figuresNumerator;
    } Numerator6;

    lookup Numerator7 {
        sub @figures' @figuresNumerator @figuresNumerator @figuresNumerator @figuresNumerator @figuresNumerator
         @figuresNumerator fraction by @figuresNumerator;
    } Numerator7;

    lookup Numerator8 {
        sub @figures' @figuresNumerator @figuresNumerator @figuresNumerator @figuresNumerator @figuresNumerator
         @figuresNumerator @figuresNumerator fraction by @figuresNumerator;
    } Numerator8;

    lookup Numerator9 {
        sub @figures' @figuresNumerator @figuresNumerator @figuresNumerator @figuresNumerator @figuresNumerator
         @figuresNumerator @figuresNumerator @figuresNumerator fraction by @figuresNumerator;
    } Numerator9;

    lookup Numerator10 {
        sub @figures' @figuresNumerator @figuresNumerator @figuresNumerator @figuresNumerator @figuresNumerator
         @figuresNumerator @figuresNumerator @figuresNumerator @figuresNumerator fraction by @figuresNumerator;
    } Numerator10;

    lookup Denominator {
        sub [fraction @figuresDenominator] @figures' by @figuresDenominator;
    } Denominator;

    sub @figures space' @figuresNumerator by uni2009;
} frac;
```

(Some Lines have been wrapped in order to fit page.)

Numerators

The `numr` feature is designed to convert all numbers to numerators.

```
feature numr {
	sub @figures by @figuresNumerator;
} numr;
```

Denominators

The `dnom` feature is designed to convert all numbers to denominators.

```
feature dnom {
	sub @figures by @figuresDenominator;
} dnom;
```

Superscript

The `sups` feature is for superscript forms.

```
feature sups {
	sub @figures by @figuresSuperscript;
	sub @letters by @lettersSuperscript;
} sups;
```

Subscript

The `subs` feature is for subscript forms.

```
feature subs {
	sub @figures by @figuresSubscript;
	sub @letters by @lettersSubscript;
} subs;
```

Figures

If your font only includes one figure style, you don't need to do anything. If you do have more than one, you have to do some awkward things due to some odd behaviors in various applications. First off, it's best to define a feature for your default figures even though it will never be used. For example, in the demo font the default figures are lining and there are old style figures as alternates. First up we need to define the lining figures feature (`lnum`) even though it will never actually be used. Then we define the old style feature (`onum`).

```
feature lnum {
	sub @figuresOldStyle by @figures;
} lnum;

feature onum {
	sub @figures by @figuresOldStyle;
} onum;
```

Likewise, if your default figures are proportional and you have tabular alternates, you need to define the proportional figures feature (`pnum`) and then define the tabular figures feature (`tnum`).

```
feature pnum {
	sub @figuresTabular by @figures;
	sub @figuresOldStyleTabular by @figuresOldStyle;
} pnum;

feature tnum {
	sub @figures by @figuresTabular;
	sub @figuresOldStyle by @figuresOldStyleTabular;
} tnum;
```

Small Caps

There are two features that invoke small caps: small caps `smcp` and "caps to small caps" or "all small caps" `c2sc` . The latter version is for situations in which the user wants everything possible, not just letters, to be converted to small cap forms.

```
feature smcp {
	sub @lowercase by @smallCaps;
} smcp;

feature c2sc {
	sub @uppercase by @smallCaps;
	sub @lowercase by @smallCaps;
	sub @figures by @figuresSmallCap;
} c2sc;
```

All Caps

There are two features that should be activated when the user indicates that they want all text converted to uppercase. The `case` feature transforms any glyphs that should be changed to an uppercase alternate. You should not define the transformation from lowercase to uppercase for alphabetic forms. The layout engine will do that for you.

```
feature case {
	sub @punctuationUppercaseOff by @punctuationUppercaseOn;
} case;
```

The `cpsp` feature allows you to define specific spacing for all caps settings:

```
feature cpsp {
	pos @uppercase <100 0 200 0>;
	pos @punctuationUppercaseOn <100 0 200 0>;
} cpsp;
```

Note that these features will not be activated whenever the user types in all capitals. The features must be activated manually.

Swashes

There are two swash features `swsh` and `cswh` . I prefer to use the contextual version `cswh` . What this feature does will depend on the swash glyphs in your font.

```
feature cswh {
      ignore sub @filled @swashInitialsOff';
      sub @swashInitialsOff' by @swashInitialsOn;
} cswh;
```

Titling Alternates

The titling alternates feature `titl` is, as its name suggests, for titling specific alternates.

```
feature titl {
      sub @uppercase by @titlingCaps;
} titl;
```

Stylistic Sets

There are 20 special feature tags that allow you to develop your own behavior that doesn't neatly fit into any of the registered layout tags. These are known as stylistic sets and have tags `ss01` , `ss02` , `ss03` and so on. The implementation of the rules in these is completely arbitrary.

Ligatures

There are several features that are designed to work with ligatures. The two most prominent are Common Ligatures `liga` and Discretionary Ligatures `dlig` .

The Common Ligatures feature is for ligatures that you think should be used almost all of the time.

```
feature liga {
      sub f i by f_i;
      sub f l by f_l;
} liga;
```

The Discretionary Ligatures feature is for ligatures that you think should be used sparingly.

```
feature dlig {
      sub 0 0 by 0_0;
} dlig;
```

As everywhere else, the ordering of the rules is very important within these features. You should always order them from longest target sequence to shortest. For example:

```
sub o f f i by o_f_f_i;
sub f f i by f_f_i;
sub f f by f_f;
sub f i by f_i;
```

Would this sequence be ordered from bottom to top, the `f_f_i` and `o_f_f_i` would have no chance of ever occurring, since their source glyphs already would have been converted beforehand.

Manual Alternate Access

There is one special feature that is used to determine the alternates displayed in the glyph access palette in popular design software. This feature `aalt` is processed after all other substitution features regardless of where you have it ordered in your feature definitions.
There are a couple of different ways to define the rules in this feature, but I prefer to do it manually with individual one from many rules.

```
feature aalt {
    sub A from [A.swash A.title A.random1 A.random2 A.sc];
    sub B from [B.swash B.title B.random1 B.random2 B.sc];
    sub C from [C.swash C.title C.random1 C.random2 C.sc];
    sub D from [D.swash D.title D.random1 D.random2 D.sc];
    sub E from [E.swash E.title E.random1 E.random2 E.sc];
    sub F from [F.swash F.title F.random1 F.random2 F.sc];
    sub G from [G.swash G.title G.random1 G.random2 G.sc];
    sub H from [H.swash H.title H.random1 H.random2 H.sc];
    sub I from [I.swash I.title I.random1 I.random2 I.sc];
    sub J from [J.swash J.title J.random1 J.random2 J.alt J.scalt J.sc];
    sub K from [K.swash K.title K.random1 K.random2 K.sc];
    sub L from [L.swash L.title L.random1 L.random2 L.sc];
    sub M from [M.swash M.title M.random1 M.random2 M.sc];
    sub N from [N.swash N.title N.random1 N.random2 N.sc];
    sub O from [O.swash O.title O.random1 O.random2 O.sc];
    sub P from [P.swash P.title P.random1 P.random2 P.sc];
    sub Q from [Q.swash Q.title Q.random1 Q.random2 Q.sc];
    sub R from [R.swash R.title R.random1 R.random2 R.sc];
    sub S from [S.swash S.title S.random1 S.random2 S.sc];
    sub T from [T.swash T.title T.random1 T.random2 T.sc];
    sub U from [U.swash U.title U.random1 U.random2 U.sc];
    sub V from [V.swash V.title V.random1 V.random2 V.sc];
    sub W from [W.swash W.title W.random1 W.random2 W.sc];
    sub X from [X.swash X.title X.random1 X.random2 X.sc];
    sub Y from [Y.swash Y.title Y.random1 Y.random2 Y.sc];
    sub Z from [Z.swash Z.title Z.random1 Z.random2 Z.sc];
    sub IJ from [IJ.dutch];
    sub zero from [zero.den zero.num zero.old zero.sub zero.sup zero.tab zero.oldtab zero.sc];
    sub one from [one.den one.num one.old one.sub one.sup one.tab one.oldtab one.sc];
    sub two from [two.den two.num two.old two.sub two.sup two.tab two.oldtab two.sc];
    sub three from [three.den three.num three.old three.sub three.sup three.tab three.oldtab three.sc];
    sub four from [four.den four.num four.old four.sub four.sup four.tab four.oldtab four.sc];
    sub five from [five.den five.num five.old five.sub five.sup five.tab five.oldtab five.sc];
    sub six from [six.den six.num six.old six.sub six.sup six.tab six.oldtab six.sc];
    sub seven from [seven.den seven.num seven.old seven.sub seven.sup seven.tab seven.oldtab seven.sc];
    sub eight from [eight.den eight.num eight.old eight.sub eight.sup eight.tab eight.oldtab eight.sc];
    sub nine from [nine.den nine.num nine.old nine.sub nine.sup nine.tab nine.oldtab nine.sc];
    sub exclamdown from [exclamdown.uc];
    sub questiondown from [questiondown.uc];
    sub at from [at.uc];
} aalt;
```

Fun Stuff

Small caps and swashes are fun, but for me the real fun is in making fonts do unexpected things. Below are various unusual problems that I've been faced with and how I solved them.

Randomization

Everyone wants their font to look like the glyphs were randomly drawn. But, let's establish something first: No one will ever do randomization better than **LettError** did in their famous **Beowolf**. **No one**. Still want try some randomization? Okay.

Randomization is a bit of a Holy Grail in the OpenType world. The problem is that it's not actually possible for a couple of reasons. For one thing, we can only select from alternates, not actually modify glyph outlines. For another, for true pseudo-randomization there needs to be an external source that influences the random selection process and we can't build a **random seed** generator with the OpenType tables. So, we have to fake it. There are a number of methods that can be used to do this. I have three that I like.

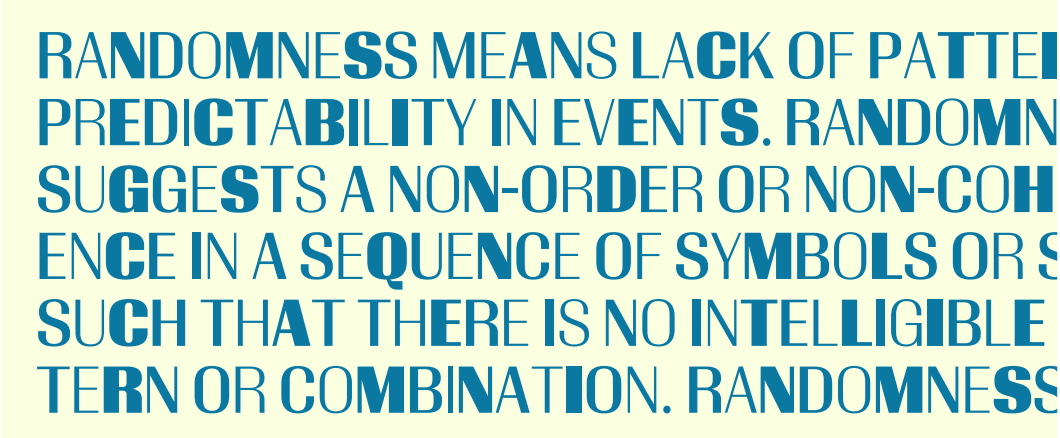
(PS: That "random alternates" feature in the OpenType Layout Tag Registry? It's not supported widely, if at all. Sorry.)

Method 1: Endless Cycle

This method is useful when you don't have a preferred version of a glyph. For example, say you draw three glyphs for every character and you want those spread out across the text. This method will cycle between the glyphs.

For example:

Given enough alternates, this creates an effective illusion in a large block of text.



The code is very simple:

```
feature calt {
    @randomCycle1 = [@uppercase];
    @randomCycle2 = [A.random1 B.random1 C.random1 D.random1 E.random1 F.random1 G.random1 H.random1 I.random1
                     J.random1 K.random1 L.random1 M.random1 N.random1 O.random1 P.random1 Q.random1 R.random1
                     S.random1 T.random1 U.random1 V.random1 W.random1 X.random1 Y.random1 Z.random1];
    @randomCycle3 = [A.random2 B.random2 C.random2 D.random2 E.random2 F.random2 G.random2 H.random2 I.random2
                     J.random2 K.random2 L.random2 M.random2 N.random2 O.random2 P.random2 Q.random2 R.random2
                     S.random2 T.random2 U.random2 V.random2 W.random2 X.random2 Y.random2 Z.random2];

    sub @randomCycle1 @randomCycle1' by @randomCycle2;
    sub @randomCycle2 @randomCycle1' by @randomCycle3;
} calt;
```

(Some Lines have been wrapped in order to fit page.)

Method 2: Duplicate Eliminator

This method is useful if you want to tightly control the deployment of alternates. Specifically, if you only want an alternate to appear if the glyph that it is an alternate to has appeared recently.

For example:

The code is lengthy, but fairly straightforward:

```
feature calt {

    @randomDuplicateSkip = [@uppercase];

    lookup RandomDuplicate1 {
        sub A A' by A.random1;
        sub B B' by B.random1;
        sub C C' by C.random1;
        sub D D' by D.random1;
        sub E E' by E.random1;
        sub F F' by F.random1;
        sub G G' by G.random1;
        sub H H' by H.random1;
        sub I I' by I.random1;
        sub J J' by J.random1;
        sub K K' by K.random1;
        sub L L' by L.random1;
        sub M M' by M.random1;
        sub N N' by N.random1;
        sub O O' by O.random1;
        sub P P' by P.random1;
        sub Q Q' by Q.random1;
        sub R R' by R.random1;
        sub S S' by S.random1;
        sub T T' by T.random1;
        sub U U' by U.random1;
        sub V V' by V.random1;
        sub W W' by W.random1;
        sub X X' by X.random1;
        sub Y Y' by Y.random1;
        sub Z Z' by Z.random1;
    } RandomDuplicate1;

    lookup RandomDuplicate2 {
        sub A @randomDuplicateSkip A' by A.random1;
        sub B @randomDuplicateSkip B' by B.random1;
        sub C @randomDuplicateSkip C' by C.random1;
        sub D @randomDuplicateSkip D' by D.random1;
        sub E @randomDuplicateSkip E' by E.random1;
        sub F @randomDuplicateSkip F' by F.random1;
        sub G @randomDuplicateSkip G' by G.random1;
        sub H @randomDuplicateSkip H' by H.random1;
        sub I @randomDuplicateSkip I' by I.random1;
        sub J @randomDuplicateSkip J' by J.random1;
        sub K @randomDuplicateSkip K' by K.random1;
        sub L @randomDuplicateSkip L' by L.random1;
        sub M @randomDuplicateSkip M' by M.random1;
        sub N @randomDuplicateSkip N' by N.random1;
        sub O @randomDuplicateSkip O' by O.random1;
        sub P @randomDuplicateSkip P' by P.random1;
        sub Q @randomDuplicateSkip Q' by Q.random1;
        sub R @randomDuplicateSkip R' by R.random1;
        sub S @randomDuplicateSkip S' by S.random1;
        sub T @randomDuplicateSkip T' by T.random1;
        sub U @randomDuplicateSkip U' by U.random1;
        sub V @randomDuplicateSkip V' by V.random1;
        sub W @randomDuplicateSkip W' by W.random1;
        sub X @randomDuplicateSkip X' by X.random1;
        sub Y @randomDuplicateSkip Y' by Y.random1;
        sub Z @randomDuplicateSkip Z' by Z.random1;
    } RandomDuplicate2;

    lookup RandomDuplicate3 {
        sub A @randomDuplicateSkip @randomDuplicateSkip A' by A.random1;
        sub B @randomDuplicateSkip @randomDuplicateSkip B' by B.random1;
        sub C @randomDuplicateSkip @randomDuplicateSkip C' by C.random1;
        sub D @randomDuplicateSkip @randomDuplicateSkip D' by D.random1;
        sub E @randomDuplicateSkip @randomDuplicateSkip E' by E.random1;
        sub F @randomDuplicateSkip @randomDuplicateSkip F' by F.random1;
        sub G @randomDuplicateSkip @randomDuplicateSkip G' by G.random1;
        sub H @randomDuplicateSkip @randomDuplicateSkip H' by H.random1;
        sub I @randomDuplicateSkip @randomDuplicateSkip I' by I.random1;
        sub J @randomDuplicateSkip @randomDuplicateSkip J' by J.random1;
        sub K @randomDuplicateSkip @randomDuplicateSkip K' by K.random1;
        sub L @randomDuplicateSkip @randomDuplicateSkip L' by L.random1;
        sub M @randomDuplicateSkip @randomDuplicateSkip M' by M.random1;
        sub N @randomDuplicateSkip @randomDuplicateSkip N' by N.random1;
        sub O @randomDuplicateSkip @randomDuplicateSkip O' by O.random1;
        sub P @randomDuplicateSkip @randomDuplicateSkip P' by P.random1;
        sub Q @randomDuplicateSkip @randomDuplicateSkip Q' by Q.random1;
        sub R @randomDuplicateSkip @randomDuplicateSkip R' by R.random1;
        sub S @randomDuplicateSkip @randomDuplicateSkip S' by S.random1;
        sub T @randomDuplicateSkip @randomDuplicateSkip T' by T.random1;
        sub U @randomDuplicateSkip @randomDuplicateSkip U' by U.random1;
        sub V @randomDuplicateSkip @randomDuplicateSkip V' by V.random1;
        sub W @randomDuplicateSkip @randomDuplicateSkip W' by W.random1;
        sub X @randomDuplicateSkip @randomDuplicateSkip X' by X.random1;
        sub Y @randomDuplicateSkip @randomDuplicateSkip Y' by Y.random1;
        sub Z @randomDuplicateSkip @randomDuplicateSkip Z' by Z.random1;
    } RandomDuplicate3;

} calt;
```

Method 3: Quantum

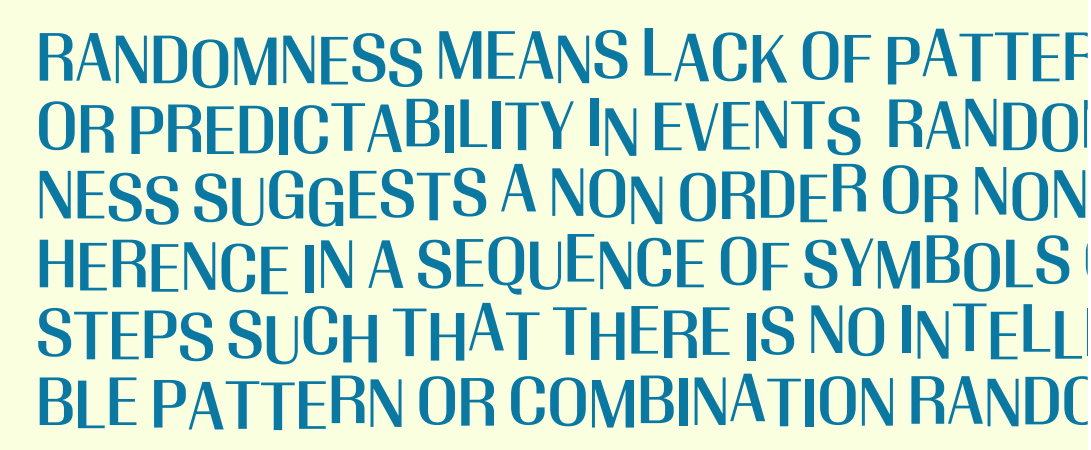
This method is for dedicated randomization aficionados. A little history about how this came about: in 2005 I wanted to see if I could come up with a randomization technique that produced less predictable results than the methods above. I realized that I could use the text that the feature is transforming as a pseudo-pseudo-random seed. I was reading a tiny bit about quantum mechanics around the same time and my very limited understanding of some of the experiments in that field gave me an idea. That's a story that I won't get into here. Anyway, the code is long and convoluted, but the idea is pretty simple. I'll give you an overview.

Before we get to rules, we establish several important classes. First we create two *trigger* classes. The first will contain half of the glyphs in the font and the next will contain the other half of the glyphs. These need to be randomly chosen. In other words, don't put A-Z in the same class. That won't produce the desired result. Next, we establish *alternate states* for glyphs. These states are defined in a series of classes. Each of these classes contain a glyph and its alternate. The next class contains the opposite of the previous class. For example:

```
@class1 = [A A.alt];
@class2 = [A.alt A];
```

Finally, we establish a *skip* class that contains everything in the font.

The glyph processing happens in a series of lookups that each pass over the entire glyph run. As a glyph, let's call it `P`, is being processed the lookup backtracks a specific number of glyphs. The glyph at the beginning of that backtrack, let's call it `B`, is then tested against a class, let's call it `@T`. Importantly, `@T` only contains half of the glyphs in the font. If `B` is in `@T`, `P` is switched to an alternate state. Thus, each glyph state is dependent on the state of all the glyphs that precede it. Given that the text is most likely going to have unpredictable letter combinations, we get a fairly effective randomization. For example:



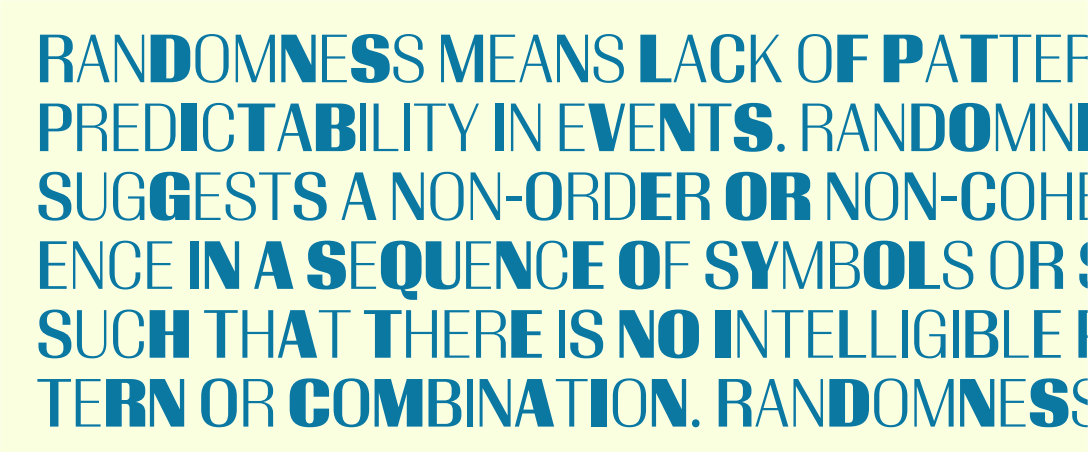
Is it real randomization? No. Is it perfect? No. Is it incredibly complex and hard to write? Yes. (Unless you use a script to write it.) Can it be slow if a font contains a large number of glyphs and the glyph run being processed is very long? Yes. Is it awesome anyway? I think so.

(Code is on the following page.)

```
feature calt {
    @randomQuantumTrigger1 = [A.random1 A.random2 B B.random2 C C.random1 C.random2 D D.random1 E.random1 G
                              G.random2 H H.random2 I J K.random1 L.random2 N.random1 O O.random1 P.random1
                              P.random2 Q.random1 S S.random1 S.random2 T T.random1 U.random2 V W.random1 W.random2
                              X X.random1 Y Y.random1 Y.random2 Z.random2];
    @randomQuantumTrigger2 = [A B.random1 D.random2 E E.random2 F F.random1 F.random2 G.random1 H.random1 I.random1
                              I.random2 J.random1 J.random2 K K.random2 L L.random1 M M.random1 M.random2 N
                              N.random2 O.random2 P Q Q.random2 R R.random1 R.random2 T.random2 U U.random1
                              V.random1 V.random2 W X.random2 Z Z.random1 space];

    @randomQuantumGlyphs1 = [A         B         C         D         E         F         G         H         I
                             J         K         L         M         N         O         P         Q         R
                             S         T         U         V         W         X         Y         Z];
    @randomQuantumGlyphs2 = [A.random1 B.random1 C.random1 D.random1 E.random1 F.random1 G.random1 H.random1
                             I.random1 J.random1 K.random1 L.random1 M.random1 N.random1 O.random1 P.random1
                             Q.random1 R.random1 S.random1 T.random1 U.random1 V.random1 W.random1 X.random1
                             Y.random1 Z.random1];
    @randomQuantumGlyphs3 = [A.random2 B.random2 C.random2 D.random2 E.random2 F.random2 G.random2 H.random2
                             I.random2 J.random2 K.random2 L.random2 M.random2 N.random2 O.random2 P.random2
                             Q.random2 R.random2 S.random2 T.random2 U.random2 V.random2 W.random2 X.random2
                             Y.random2 Z.random2];

    @randomQuantumState1 = [@randomQuantumGlyphs1 @randomQuantumGlyphs3 @randomQuantumGlyphs2];
    @randomQuantumState2 = [@randomQuantumGlyphs2 @randomQuantumGlyphs1 @randomQuantumGlyphs3];
    @randomQuantumState3 = [@randomQuantumGlyphs3 @randomQuantumGlyphs2 @randomQuantumGlyphs1];

    @randomQuantumSkip = [@uppercase space];

    lookup RandomQuantum10 {
        sub @randomQuantumTrigger1 @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip
         @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip
         @randomQuantumSkip @randomQuantumSkip @randomQuantumState1' by @randomQuantumState2;
    } RandomQuantum10;

    lookup RandomQuantum9 {
        sub @randomQuantumTrigger2 @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip
         @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip
         @randomQuantumState2' by @randomQuantumState3;
    } RandomQuantum9;

    lookup RandomQuantum8 {
        sub @randomQuantumTrigger1 @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip
         @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumState3' by
         @randomQuantumState1;
    } RandomQuantum8;

    lookup RandomQuantum7 {
        sub @randomQuantumTrigger2 @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip
         @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumState1' by @randomQuantumState2;
    } RandomQuantum7;

    lookup RandomQuantum6 {
        sub @randomQuantumTrigger1 @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip
         @randomQuantumSkip @randomQuantumSkip @randomQuantumState2' by @randomQuantumState3;
    } RandomQuantum6;

    lookup RandomQuantum5 {
        sub @randomQuantumTrigger2 @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip
         @randomQuantumSkip @randomQuantumState3' by @randomQuantumState1;
    } RandomQuantum5;

    lookup RandomQuantum4 {
        sub @randomQuantumTrigger1 @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip
         @randomQuantumState1' by @randomQuantumState2;
    } RandomQuantum4;

    lookup RandomQuantum3 {
        sub @randomQuantumTrigger2 @randomQuantumSkip @randomQuantumSkip @randomQuantumSkip @randomQuantumState2'
         by @randomQuantumState3;
    } RandomQuantum3;

    lookup RandomQuantum2 {
        sub @randomQuantumTrigger1 @randomQuantumSkip @randomQuantumSkip @randomQuantumState3' by
         @randomQuantumState1;
    } RandomQuantum2;

    lookup RandomQuantum1 {
        sub @randomQuantumTrigger2 @randomQuantumState1' by @randomQuantumState2;
    } RandomQuantum1;

    lookup RandomQuantum0 {
        sub @randomQuantumTrigger1 @randomQuantumState2' by @randomQuantumState3;
    } RandomQuantum0;

} calt;
```

Bonus: Quantum Positioning

It's possible to extend the quantum randomization method above and use it to randomly shift glyphs around.



Possible, but probably not advisable. The support for this is almost certainly going to be uneven.

The code is similar to the substitution method. In this case, instead of states, the cumulative effect that lookups have on glyph records is used.

(Code is on the following page.)

```
feature ss01 {
    @randomPositionQuantumTrigger1 = [A.random2 B B.random1 C.random2 D D.random1 E.random2 F F.random1
                                      F.random2 G G.random2 H H.random2 I.random1 I.random2 J.random1 J.random2
                                      K.random2 L.random1 M.random1 N O Q Q.random1 R S S.random1 T.random1
                                      T.random2 U U.random1 U.random2 W.random1 X.random1 X.random2 Z Z.random2
                                      space];
    @randomPositionQuantumTrigger2 = [A A.random1 B.random2 C C.random1 D.random2 E E.random1 G.random1
                                      H.random1 I J K K.random1 L L.random2 M M.random2 N.random1 N.random2
                                      O.random1 O.random2 P P.random1 P.random2 Q.random2 R.random1 R.random2
                                      S.random2 T V V.random1 V.random2 W W.random2 X Y Y.random1 Y.random2
                                      Z.random1];

    @randomPositionQuantumTarget = [@randomPositionQuantumTrigger1 @randomPositionQuantumTrigger2];

    @randomPositioningQuantumSkip = [@randomPositionQuantumTrigger1 @randomPositionQuantumTrigger2];

    lookup RandomPositioningQuantum9 {
        pos @randomQuantumTrigger2 @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositioningQuantumSkip @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositioningQuantumSkip @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositioningQuantumSkip @randomPositionQuantumTarget' <0 50 0 0>;
    } RandomPositioningQuantum9;

    lookup RandomPositioningQuantum8 {
        pos @randomQuantumTrigger1 @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositioningQuantumSkip @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositioningQuantumSkip @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositionQuantumTarget' <0 40 0 0>;
    } RandomPositioningQuantum8;

    lookup RandomPositioningQuantum7 {
        pos @randomQuantumTrigger2 @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositioningQuantumSkip @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositioningQuantumSkip @randomPositioningQuantumSkip @randomPositionQuantumTarget'
            <0 30 0 0>;
    } RandomPositioningQuantum7;

    lookup RandomPositioningQuantum6 {
        pos @randomQuantumTrigger1 @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositioningQuantumSkip @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositioningQuantumSkip @randomPositionQuantumTarget' <0 20 0 0>;
    } RandomPositioningQuantum6;

    lookup RandomPositioningQuantum5 {
        pos @randomQuantumTrigger2 @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositioningQuantumSkip @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositionQuantumTarget' <0 10 0 0>;
    } RandomPositioningQuantum5;

    lookup RandomPositioningQuantum4 {
        pos @randomQuantumTrigger1 @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositioningQuantumSkip @randomPositioningQuantumSkip @randomPositionQuantumTarget'
            <0 -10 0 0>;
    } RandomPositioningQuantum4;

    lookup RandomPositioningQuantum3 {
        pos @randomQuantumTrigger2 @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositioningQuantumSkip @randomPositionQuantumTarget' <0 -20 0 0>;
    } RandomPositioningQuantum3;

    lookup RandomPositioningQuantum2 {
        pos @randomQuantumTrigger1 @randomPositioningQuantumSkip @randomPositioningQuantumSkip
            @randomPositionQuantumTarget' <0 -30 0 0>;
    } RandomPositioningQuantum2;

    lookup RandomPositioningQuantum1 {
        pos @randomQuantumTrigger2 @randomPositioningQuantumSkip @randomPositionQuantumTarget' <0 -40 0 0>;
    } RandomPositioningQuantum1;

    lookup RandomPositioningQuantum0 {
        pos @randomQuantumTrigger1 @randomPositionQuantumTarget' <0 -50 0 0>;
    } RandomPositioningQuantum0;

} ss01;
```

6 | Style Guide

I like to make my code look nice. I'm a designer, so I'll even go so far as to say that I try to make it look typographic. My goal is to make it comfortable to read because, as **Guido van Rossum** once put it, code is read much more often than it is written. Debugging problems is not fun. Debugging problems in poorly structured, hard to read code is a bag of hurt.

Honestly, my feelings won't be hurt if you don't use my style recommendations. There is plenty in this style guide to disagree with and even I don't always follow these rules. What I do hope you will do is find a consistent way to structure and style your code so that it is easy for you to work with. That said, I am 100% right and everyone else is 100% wrong.

My preferred style has grown out of my experience with Python and my general adherence to the legendary **PEP 8**. So, it shares some traits with Python code.

Whitespace

I use whitespace, both horizontal and vertical, to indicate logical groupings. Indentation is used to visually connect features, lookups, scripts and languages. Blank lines are used to indicate feature changes, sections and so on.

I prefer that indentation be done with spaces (four per indentation level), but I'm not religious about it. If you prefer tabs, use tabs. However, **do not ever mix tabs and spaces**. Similarly, do not put pointless spaces at the end of a line.

There should be no lines that contain no characters other than spaces. That's just gross.

Column Width

I don't have a set column width for class definitions or rules. I find that they are harder to read if they are broken over multiple lines than if they are in one long line. I do wrap my comments if they are longer than 50-60 characters in length. I try to follow standard typographic practice when writing and breaking lines in comments. Comments are among the most important things in the code, so I try to make them ultra readable.

Comments

I precede all comments with a single # followed by a space.

```
# This looks nice.
#This does not.
```

Comments are always indented to the level of the code that they are meant to document. Sudden horizontal shifts are jarring.

I like to use proper title and sentence casing in comments with occasional all uppercase for EMPHASIS. if you are an e e cummings fan, feel free to use all lowecase.

Features

I structure my features like this:

```
# -----------------
# Example Feature 1
# -----------------

feature xmp1 {
      # rules, lookups, etc.
} xmp1;


# -----------------
# Example Feature 2
# -----------------

feature xmp2 {
      # rules, lookups, etc.
} xmp2;
```

Each feature, or group of features in the case of interrelated features like figure styles, gets a human readable header with line of hyphens above and below. The number of hyphens matches the number of characters in the header. One blank line follows this.

The feature begins unindented with the keyword, a space, the tag, a space and the opening brace. It is closed with an unindented closing brace, a space, the tag and the closing semicolon. Between the opening and closing lines, the indentation level of any line containing text is increased one level.

Features always have two blank lines between them and whatever is next in the file.

Lookups

Lookups are structured like this:

```
lookup ExampleLookup1 {
      # rules
} ExampleLookup1;

lookup ExampleLookup2 {
      # rules
} ExampleLookup2;
```

The lookup begins at the current indent level with the keyword, a space, the lookup name, a space and the opening brace. It is closed with an unindented closing brace, a space, the lookup name and the closing semicolon. Between the opening and closing lines, the indentation level of any line containing text is increased one level.

Lookup names should be descriptive of what the lookup does. For example, `SwashInitials`, `QuadrupleLigatures`, `FractionBar`. The names are composed of characters in `A-z a-z` and `0-9`. The first letter should be capitalized and other letters may be capitalized as needed to improve readability.

There should be at least one blank line above a lookup and one blank line below.

Classes

Classes are defined line this:

```
@exampleClass1 = [A B C];
@exampleClass2 = [a b c];
```

The definition begins at the current indent level with the `@` special character, the class name, a space, an `=` , a space, the opening bracket, a space delimited list of glyph names in a meaningful order, a closing bracket and the closing semicolon.

Classes should be defined on a single line. The definition may be broken into multiple lines only in rare circumstances.

Class names should be descriptive of the class' contents and purpose. For example, `@uppercase` , `@figuresNumerator` , `@randomCycle1` . If there are two classes that are intended for substituting with each other, for example if you have swash targets and swash replacements, it's good to tag the classes with `Off` and `On` for clarity. For example, `@swashInitialsOff` and `@swashInitialsOn` . Class names are composed of characters in `A-z` `a-z` and `0-9` . The first letter should be lowercase and other letters may be capitalized as needed to improve readability.

If there are two or more classes that are intended for substituting with each other, readability can be greatly increased by using spaces between glyph names as alignment padding. For example:

```
@figures         = [zero     one     two];
@figuresTabular  = [zero.tab one.tab two.tab];
@figuresSmallCap = [zero.sc  one.sc  two.sc];
```

Rules

Rules are defined with the short version of the keywords, `sub` and `pos` , instead of the long versions, `substitution` and `position` . The rules are always on a single line and they are always written at the current indentation level. Class names are preferred over inline classes, but that's not always practical or possible. There should be no double spaces and there should be no space before the closing semicolon.

Script and Language

Script and language definitions each increase the indentation level by one level until the next script or language definition or until the end of the feature. For example:

```
feature locl {

      script latn;

            language TRK exclude_dflt;

                  lookup IDOT {
                        sub i' by idotaccent;
                  } IDOT;

            language AZE exclude_dflt;
                  lookup IDOT;

            language CRT exclude_dflt;
                  lookup IDOT;

            language ROM exclude_dflt;

                  lookup SCEDILLA {
                        sub scedilla by uni0219;
                        sub Scedilla by uni0218;
                  } SCEDILLA;

} locl;
```

Acknowledgements

Thanks to the Adobe Type Department for developing the .fea syntax. It gets criticized from time to time, but it is actually a great abstraction of the GSUB, GPOS and GDEF binary data structures. I've tried, and failed, to invent a better syntax several times. .fea is hard to beat. Thanks Adobe!

Thanks to **Ben Kiel**, **Frank Grießhammer**, **Miguel Sousa** and **Erik van Blokland** for graciously reading, editing and making suggestions about early drafts of this cookbook.

Brook Elgie built the infrastructure for the site and did most of the styling work. I am hugely thankful for his efforts because CSS and me are not friends.

Thanks to **Frederik Berlean** for many things, especially his **Pygments lexer for the .fea syntax.**

Thanks to **Mr. Ken Barber** for designing typefaces that needed bizarre feature behavior that had never existed before.

Finally, thanks to my students for being so curious that it made me finally get around to writing this cookbook.

Licenses, Copyrights, and Credits

Future Versions

September 4, 2014

Initial version.

Future Versions

Near Future

- Update the fonts used for all text to include kerning. (I'm working on it!)
- Illustrations for each feature in Common Techniques.
- Animated step through of the processing sequence in Foundation Concepts.
- Add something about the methodology for developing complex features: don't dive into code. Rather make some examples of what you want the final result to look like, find the visual patterns within that and build the code from there.

Distant Future

- Include information and examples for the additional positioning rule types.
- Add a troubleshooting section. (**See this commit for a list.**)
- Add a section detailing what isn't covered by this cookbook. (**See this commit for a list.**)