

# Tutorial



This tutorial is also available as a [pdf version](#). If you wish to follow along with the tutorial, [this bundle](#) should provide you with the basic files you need.

I shall not presume to teach aesthetics, I concentrate solely on the mechanics here.

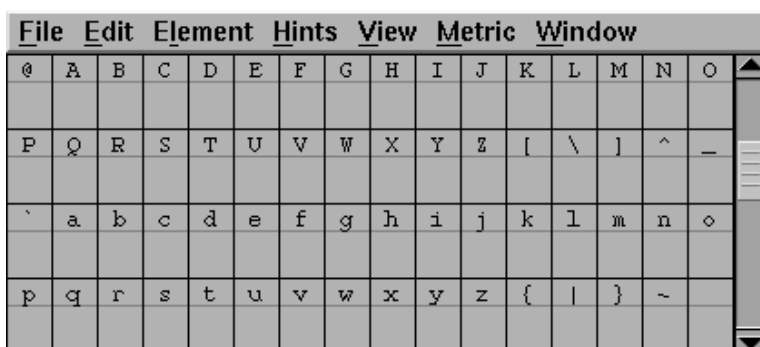
- [Font Creation](#)
- [Creating a glyph \(tracing outlines\)](#)
- [Navigating to other glyphs](#)
- [On to the next glyph \(consistent directions\)](#)
- [Consistent serifs and stem widths](#)
- [Building accented glyphs](#)
- [Building a ligature](#)
- [Lookups and features](#)
- [Examining metrics](#)
  - [Setting the baseline to baseline spacing of a font](#)
- [Kerning](#)
- [Glyph variants](#)
- [Anchoring marks](#)
- [Conditional features](#)
- [Checking your font](#)
- [Bitmaps](#)
- [Generating it](#)
- [Font Families](#)
- [Final Summary](#)
- [Bitmap strikes](#)
- [Scripting Tutorial](#)
- [Notes on various scripts](#)

**NOBLEMAN:** Now this is what I call workmanship. There is nothing on earth more exquisite than a bonny book, with well-placed columns of rich black writing in beautiful borders, and illuminated pictures cunningly inset. But nowadays, instead of looking at books, people read them. A book might as well be one of those orders for bacon and bran that you are scribbling.

-- Saint Joan, Scene IV  
George Bernard Shaw, 1924

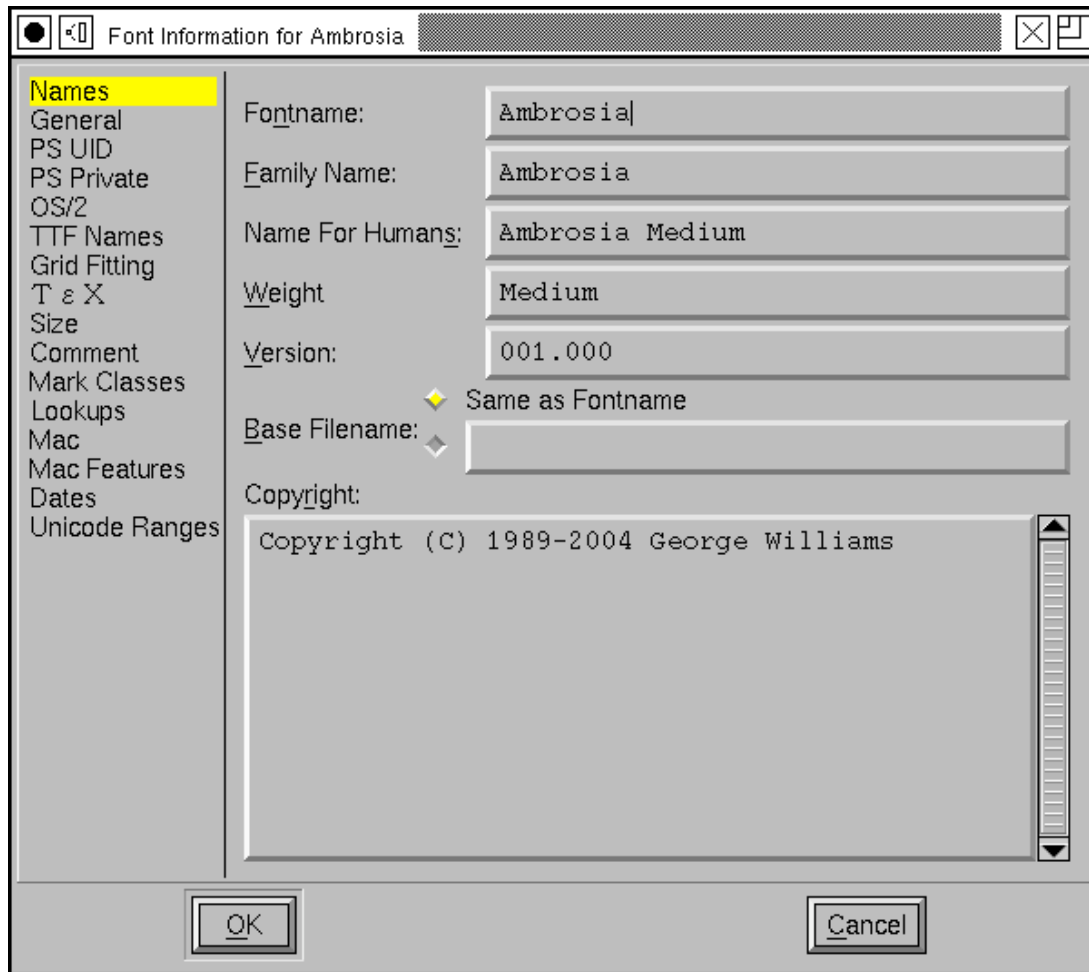
## Font creation

First create a new font with the `New` command in the `File` menu (or by using the `-new` argument at startup).



Give the font a name with the [Font Info](#) command from the `Element` menu. You use this same command to set the copyright message and change the ascent and descent (the sum of these two determines the size of the em square for the font, and by convention is 1000 for postscript fonts, a power of two (often 2048 or 4096) for truetype fonts and 15,000 for

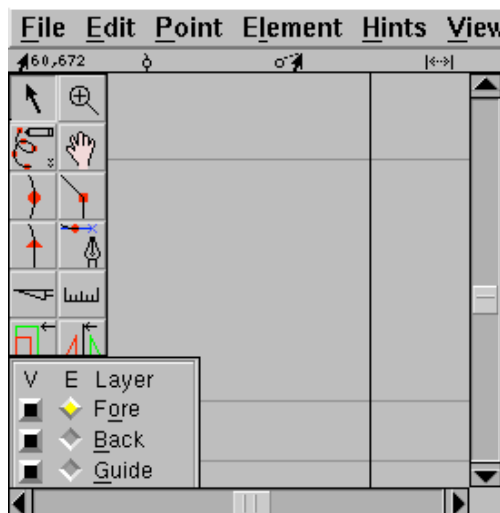
Ikarus fonts). (Also if you are planning on making a truetype font you might want to check the Quadratic Splines checkbox to use the native truetype format. Editing is a little more difficult in this mode though)



You may also wish to use Encoding->Reencode to change what characters are available in your font. FontForge generally creates new fonts with an ISO-8859-1, which contains (most of) the characters needed for Western Europe (the latin letters, some accented letters, digits, and symbols).

## Creating a glyph

Once you have done that you are ready to start editing glyphs. Double click on the entry for "C" in the font view above. You should now have an empty Outline Glyph window:

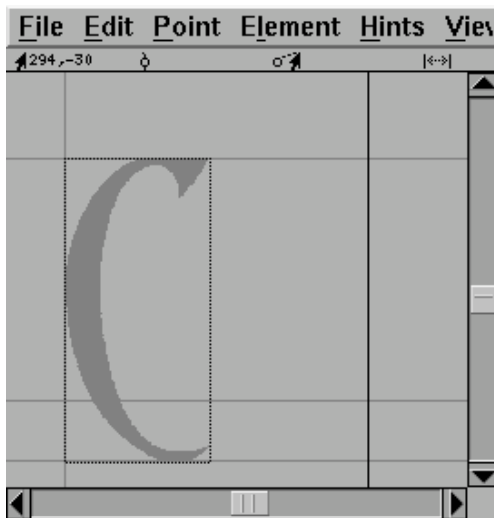


The outline glyph window contains two palettes snuggled up on the left side of the window. The top palette contains a set of editing tools, and the bottom palette controls which layers of the window are visible or editable.

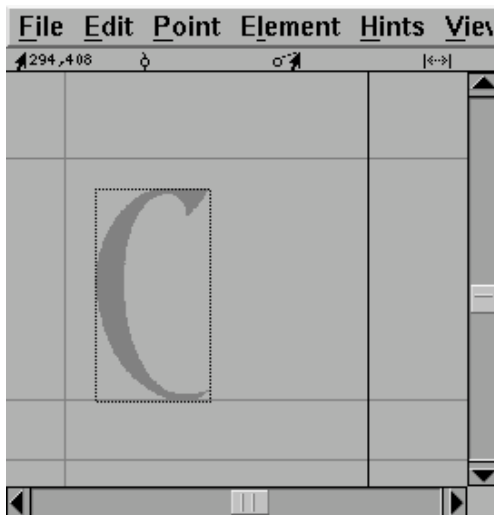
The foreground layer contains the outline that will become part of the font. The background layer can contain images or line drawings that help you draw this particular glyph. The guide layer contains lines that are useful on a font-wide basis (such as the x-height). Currently all layers are empty.

This window also shows the glyph's internal coordinate system with the x and y axes drawn in light grey. A line representing the glyph's advance width is drawn in black at the right edge of the window. FontForge assigns an advance width of one em (in PostScript that will usually be 1000 units) to the advance width of a new glyph.

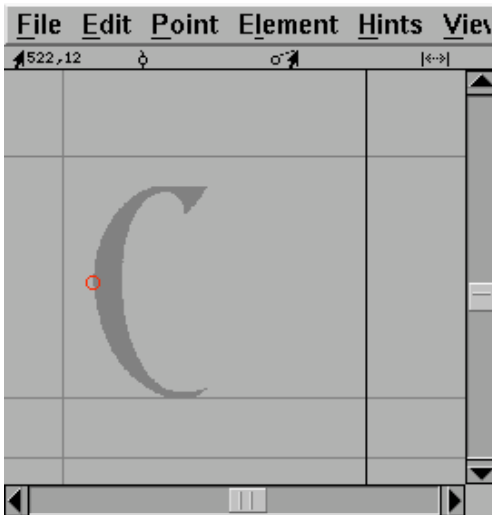
Select the Import command from the File menu and import an image of the glyph you are creating. It will be scaled so that it is as high as the em-square.



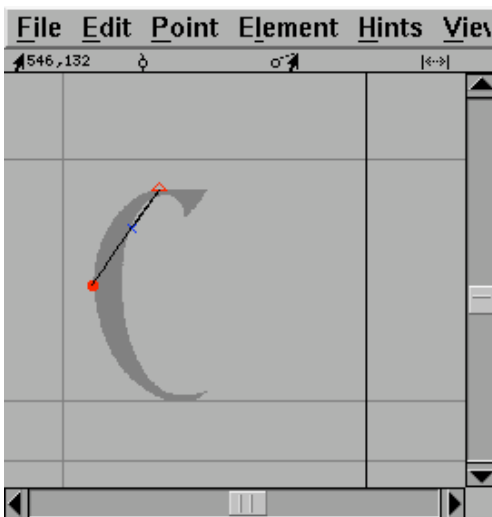
Select the background layer as editable from the layers palette, move the mouse pointer to one of the edges of the image, hold down the shift key, depress and drag the corner until the image is a reasonable size, then move the pointer onto the dark part of the image, depress the mouse and drag until the image is properly positioned.



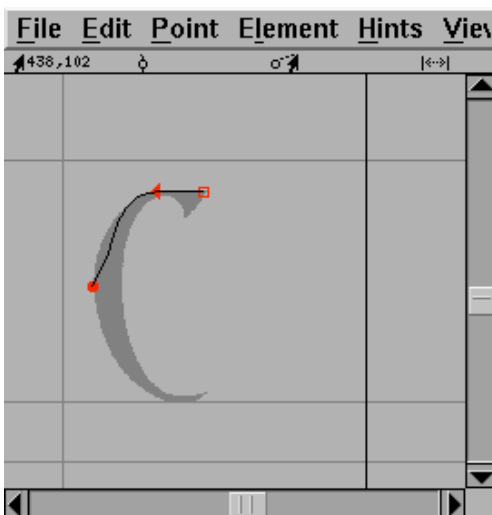
If you have downloaded the [autotrace program](#) you can invoke `Element->AutoTrace` to generate an outline from the image. But if you have not you must add points yourself. Change the active layer to be the foreground, and go to the tools palette and select the round (or curve) point. Then move the pointer to the edge of the image and add a point. I find that it is best to add points at places where the curve is horizontal or vertical, at corners, or where the curve changes inflection (A change of inflection occurs in a curve like "S" where the curve changes from being open to the left to being open on the right. If you follow these rules hinting will work better.



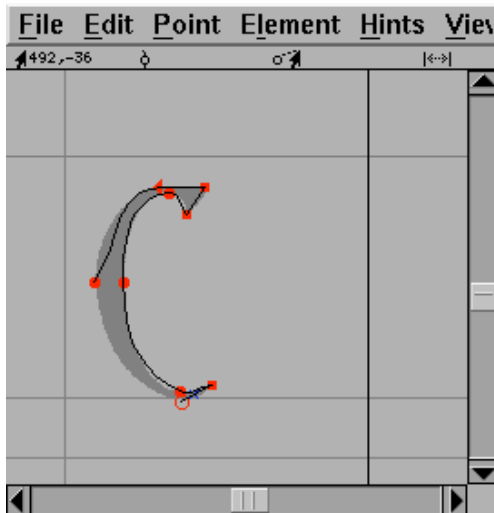
It is best to enter a curve in a clockwise fashion, so the next point should be added up at the top of the image on the flat section. Because the shape becomes flat here, a curve point is not appropriate, rather a tangent point is (this looks like a little triangle on the tools palette). A tangent point makes a nice transition from curves to straight lines because the curve leaves the point with the same slope the line had when it entered.



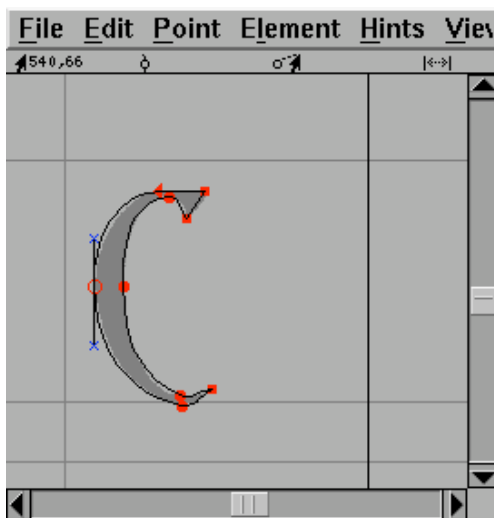
At the moment this "curve" doesn't match the image at all, don't worry about that we'll fix it later, and anyway it will change on its own as we continue. Note that we now have a control point attached to the tangent point (the little blue x). The next point needs to go where the image changes direction abruptly. Neither a curve nor a tangent point is appropriate here, instead we must use a corner point (one of the little squares on the tools palette).



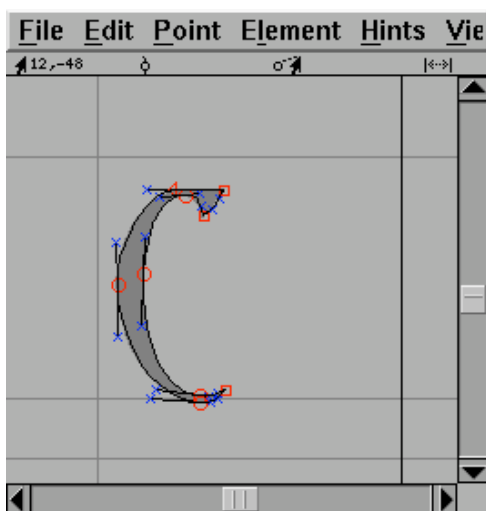
As you see the old curve now follows the image a bit more closely. We continue adding points until we are ready to close the path.



Then we close the path just by adding a new point on top of the old start point

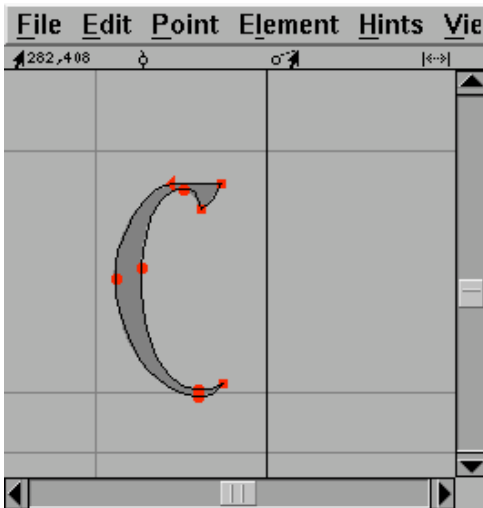


Now we must make the curve track the image more closely, to do this we must adjust the control points (the blue "x"es). To make all the control points visible select the pointer tool and double-click on the curve and then move the control points around until the curve looks right.



Finally we set width. Again with the pointer tool, move the mouse to the width line on the right edge of the screen, depress

and drag the line back to a reasonable location.



And we are done with this glyph.

If you are mathematically inclined you may be interested in the coordinates that fontforge shows in the upper left of the window. Generally you can draw glyphs quite happily without bothering about these, but for those who are interested here is some basic info:

- Each glyph has its own coordinate system.
- The vertical origin is the font's baseline (the line on which most latin letters rest)
- The horizontal origin is the place where drawing the glyph will commence. In the example above what gets drawn initially is empty space, that is fairly common, and that empty space (the distance from the origin to the left edge of the glyph) is called the left side bearing.
- The units of the coordinate system are determined by the em-size of the font. This is the sum of the font's ascent and descent. In the example above the font's ascent is 800 and descent is 200, and the ascent line (the one just above the top of the "C") is 800 units from the baseline, while the descent line is 200 units below.
- So a position of 282,408 (as above) means that the cursor is 282 units right of the horizontal origin and 408 units above the baseline (or halfway between baseline and ascent).

## Navigating to glyphs.

The font view provides one way of navigating around the glyphs in a font. Simple scroll around it until you find the glyph you need and then double click on it to open a window looking at that glyph.

Typing a glyph will move to that glyph.

However some fonts are huge (Chinese, Japanese and Korean fonts have thousands or even tens of thousands of glyphs) and scrolling around the font view is a an inefficient way of finding your glyph. `view->Goto` provides a simple dialog which will allow you to move directly to any glyph for which you know the name (or encoding). If your font is a Unicode font, then this dialog will also allow you to find glyphs by block name (ie. Hebrew rather than Alef).

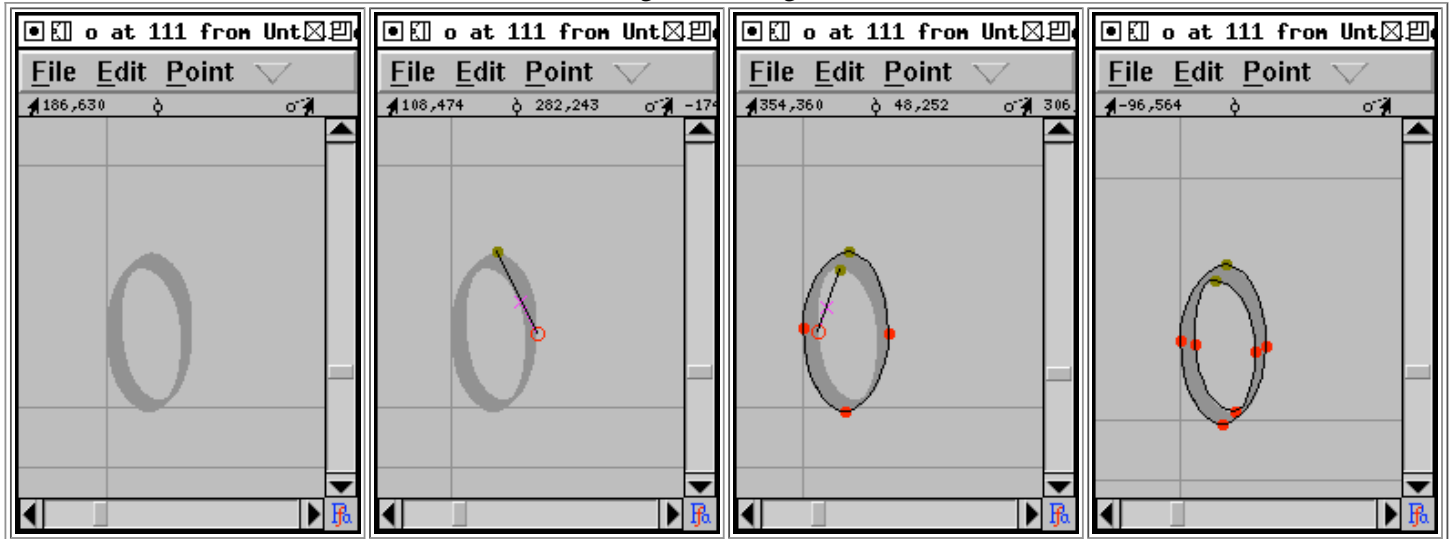
The simplest way to navigate is just to go to the next or previous glyph. And `view->Next Char` and `view->Prev Char` will do exactly that.

## Creating the letter "o" -- consistent directions


In the previous example the bitmap of the letter filled the canvas of the image. And when FontForge imported the image it needed to be scaled once in the program. But usually when you create the image of the letter you have some idea of how much white space there should be around it. If your images are exactly one em high then FontForge will scale them automatically to be the right size. So in the following examples all the images have exactly the right amount of white-space around them to fit perfectly in an em.

For the next example double click on the square in the font view that should contain "o", and import "o\_Ambrosia.png" into it.

Stages in editing "o"



Notice that the first outline is drawn clockwise and the second counter-clockwise. This change in drawing direction is important. Both PostScript and TrueType require that the outer boundary of a glyph be drawn in a certain direction (they happen to be opposite from each other, which is a mild annoyance), within FontForge all outer boundaries must be drawn clockwise, while all inner boundaries must be drawn counter-clockwise.

If you fail to alternate directions between outer and inner boundaries you may get results like the one on the left . If you fail to draw the outer contour in a clockwise fashion the errors are more subtle, but will generally result in a less pleasing result once the glyph has been rasterized.

**TECHNICAL AND CONFUSING:** the exact behavior of rasterizers varies. Early PostScript rasterizers used a "non-zero winding number rule" while more recent ones use an "even-odd" rule. TrueType uses the "non-zero" rule. The description given above is for the "non-zero" rule. The "even-odd" rule would fill the "o" correctly no matter which way the paths were drawn (though there would probably be subtle problems with hinting).

Filling using the even-odd rules that a line is drawn from the current pixel to infinity (in any direction) and the number of contour crossings is counted. If this number is even the pixel is not filled. If the number is odd the pixel is filled. In the non-zero winding number rule the same line is drawn, contour crossings in a clockwise direction add 1 to the crossing count, counter-clockwise contours subtract 1. If the result is 0 the pixel is not filled, any other result will fill it.

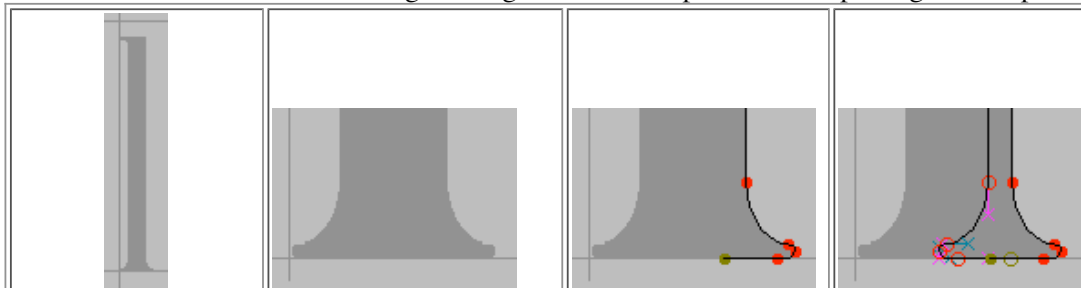
The command [Element->Correct\\_Direction](#) will look at each selected contour, figure out whether it qualifies as an outer or inner contour and will reverse the drawing direction when the contour is drawn incorrectly.

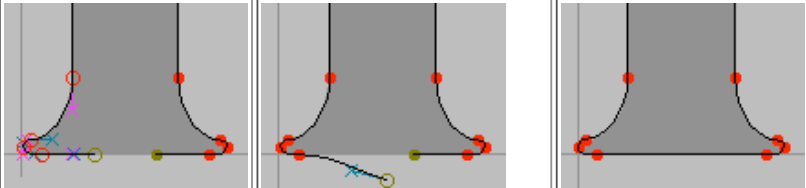
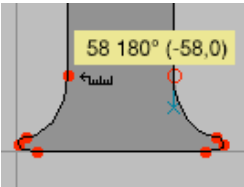

## Creating letters with consistent stem widths, serifs and heights.

Many Latin (Greek, Cyrillic) fonts have serifs, special terminators at the end of stems. And in almost all LGC fonts there should only be a small number of stem widths used (ie. the vertical stem of "l" and "i" should probably be the same).

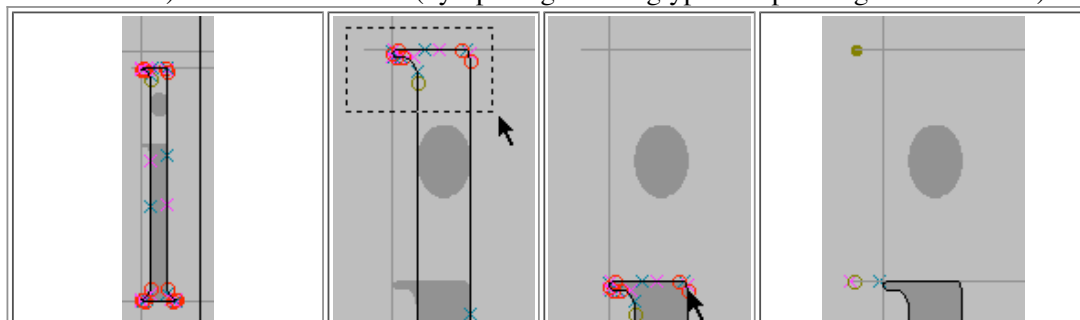
FontForge does not have a good way to enforce consistency, but it does have various commands to help you check for it, and to find discrepancies.

Let us start with the letter "l" and go through the familiar process of importing a bitmap and defining it's outline.



The imported image	Use the magnify tool to examine the bottom serif, and note that it is symmetric left to right.	Outline the right half of the serif	select the outline, invoke <code>Edit -&gt; Copy</code> then <code>Edit -&gt; Paste</code> , and finally <code>Element -&gt; Transform -&gt; Transform</code> and select <code>Flip</code> (from the pull down list) and check <code>Horizontal</code>
			
Drag the flipped serif over to the left until it snuggles up against the left edge of the glyph	Deselect the path, and select one end point and drag it until it is on top of the end point of the other half	Finish off the glyph	
			
But let's do two more things. First let's measure the stem width, and second let's mark the height of the "l"	Select the ruler tool from the tool palette, and drag it from one edge of the stem to the other. A little window pops up showing the width is 58 units, the drag direction is 180 degrees, and the drag was -58 units horizontally, and 0 units vertically.	Go to the layers palette and select the Guide radio box (this makes the guide layer editable). Then draw a line at the top of the "l", this line will be visible in all glyphs and marks the ascent height of this font.	

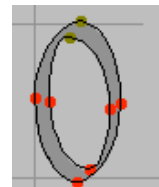
Now let's do "i". This glyph looks very much like a short "l" with a dot on top. So let's copy the "l" into the "i"; this will automatically give us the right stem width and the correct advance width. The copy may be done either from the font view (by selecting the square with the "l" in it and pressing `Edit->Copy`) or from the outline view (by `Edit->Select->Select All` and `Edit->Copy`). Similarly the Paste may be done either in the font view (by selecting the "i" square and pressing `Edit->Paste`) or the outline view (by opening the "i" glyph and pressing `Edit->Paste`).





Import the "i" image, and copy the "l" glyph.	Select the top serif of the l	drag it down to the right height	go to the guide layer and add a line at the x-height
---	-------------------------------	----------------------------------	--

Let's look briefly back at the "o" we built before. You may notice that the "o" reaches a little above the guide line we put in to mark the x-height (and a little below the baseline). This is called overshoot and is an attempt to remedy an optical illusion. A curve actually needs to rise about 3% (of its diameter) above the x-height for it to appear on the x-height.



Let's look at "k". Again we will copy an "l" into it and import an appropriate image.

Import the "k" image and copy the "l" glyph. Note that the x-height line matches the "k" (as we would hope). Also note that the width of the "l" is inappropriate for "k" so we'll have to select it and drag it over.	Select the knife tool from the palette, and cut the stem of the "l" shape at appropriate points for "k".	Remove the splines between the cut points. An easy way to do this is to grab the spline itself, (which selects its end points) and then do Edit -> Clear.
Select the end points and convert them into corner points with Point -> Corner.	Then draw in the outer contour.	And the inner contour. Finally do an Edit -> Select -> Select All and an Element -> Correct Direction.

## Building accented glyphs

Latin, Greek and Cyrillic all have a large complement of accented glyphs. FontForge provides several ways to build accented glyphs out of base glyphs.

The most obvious mechanism is simple copy and paste: [Copy](#) the letter "A" and [Paste](#) it to "Ã" then [copy](#) the tilde accent and [Paste it Into](#) "Ã" (note Paste Into is subtly different from Paste. Paste clears out the glyph before pasting, while Paste Into merges what was in the glyph with the what is in the clipboard). Then you open up "Ã" and position the accent so that it appears properly centered over the A.

This mechanism is not particularly efficient, if you change the shape of the letter "A" you will need to regenerate all the accented glyphs built from it. FontForge has the concept of a [Reference](#) to a glyph. So you can Copy a Reference to "A", and Paste it, the Copy a Reference to tilde and Paste it Into, and then again adjust the position of the accent over the A.

Then if you change the shape of the A the shape of the A in "Ã" will be updated automatically -- as will the width of "Ã".

But FontForge knows that "Ã" is built out of "A" and the tilde accent, and it can easily create your accented glyphs itself by placing the references in "Ã" and then positioning the accent over the "A". (The Unicode consortium provides a database which lists the components of every accented glyph in Unicode and FontForge uses this).

As an example, open the file: tutorial/Ambrosia.sfd, then select all the glyphs at encodings 0xc0-0xff, and then press [Element->Build->Build Accented](#) all the accented glyphs will magically appear (there are a few glyphs in this range which are not accented, and they will remain blank).

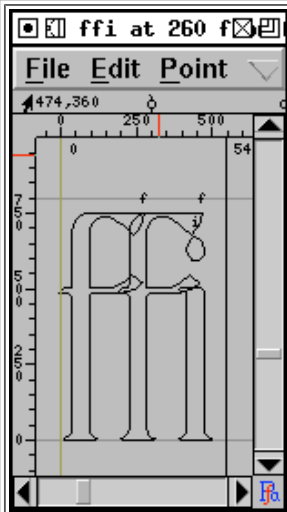
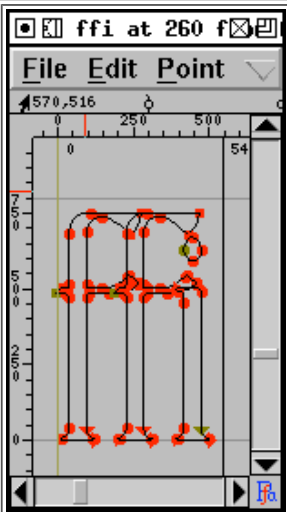
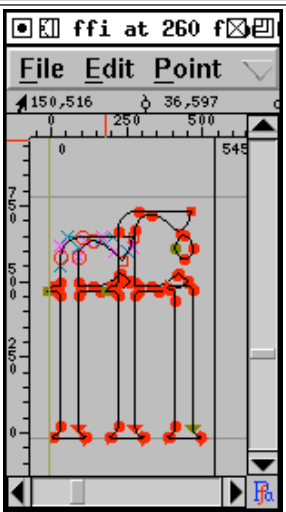
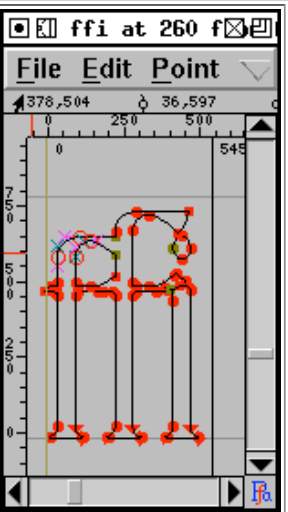
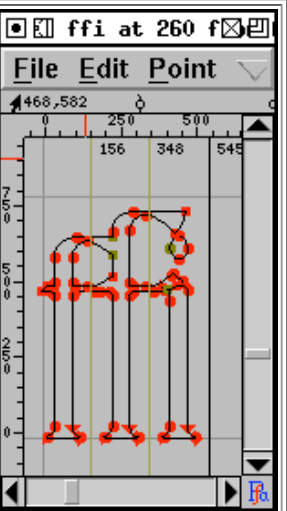
FontForge has a heuristic for positioning accents (most accents are centered over the highest point of the glyph), sometimes this will produce bad results (if the one of the two stems of "u" is slightly taller than the other the accent will be placed over it rather than being centered over the glyph), so you should be prepared to look at your accented glyphs after creating them. You may need to adjust one or two (or you may even want to redesign your base glyphs slightly).

## Creating a ligature

Unicode contains a number of ligature glyphs (in latin we have: Æ, OE, fi, etc. while in arabic there are hundreds). Again Unicode provides a database listing the components of each standard ligature.

FontForge cannot create a nice ligature for you, but what it can do is put all the components of the ligature into the glyph with [Element->Build->Build Composite](#). This makes it slightly easier (at least in latin) to design a ligature.

### Steps to building a ligature

				
Use the <a href="#">Element -&gt; Glyph Info</a> dialog to name the glyph and mark it as a ligature. Then use <a href="#">Element -&gt; Build -&gt; Build Composite</a> to insert references to the ligature components.	Use the <a href="#">Edit-&gt; Unlink References</a> command to turn the references into a set of contours.	Adjust the components so that they will look better together. Here the stem of the first f has been lowered.	Use the <a href="#">Element -&gt; Remove Overlap</a> command to clean up the glyph.	Finally drag the ligature caret lines from the origin to more appropriate places between the components.

Some word processors will allow the editing caret to be placed inside a ligature (with a caret position between each component of the ligature). This means that the user of that word processor does not need to know s/he is dealing with a ligature and sees behavior very similar to what s/he would see if the components were present. But if the word processor is to be able to do this it must have some information from the font designer giving the locations of appropriate caret positions. As soon as FontForge notices that a glyph is a ligature it will insert in it enough caret location lines to fit between the ligature's components. FontForge places these on the origin, if you leave them on the origin FontForge will ignore them. But once you have built your ligature you might want to move the pointer tool over to the origin line, press the button and drag one of the caret lines to its correct location. (Only Apple Advanced Typography and OpenType support this).

There are a good many ligatures needed for the indic scripts, but Unicode does not provide an encoding for them. If you wish to build a ligature that is not part of Unicode you may do so. First [add an unencoded glyph to your font](#) (or if your font is a Unicode font, you could use a code point in the private use area), and name the glyph. The name is important, if you

name it correctly FontForge will be able to figure out that it is a ligature and what its components are. If you want to build a ligature out of the glyphs "longs", "longs" and "l" then name it "longs\_longs\_l", if you want to build a ligature out of Unicode 0D15, 0D4D and 0D15 then name it "uni0D15\_uni0D4D\_uni0D15".

Once you have named your ligature, and inserted its components (with Build Composite), you probably want to open the glyph, [Unlink your References](#) and edit them to make a pleasing shape (as above).

## Lookups and Features

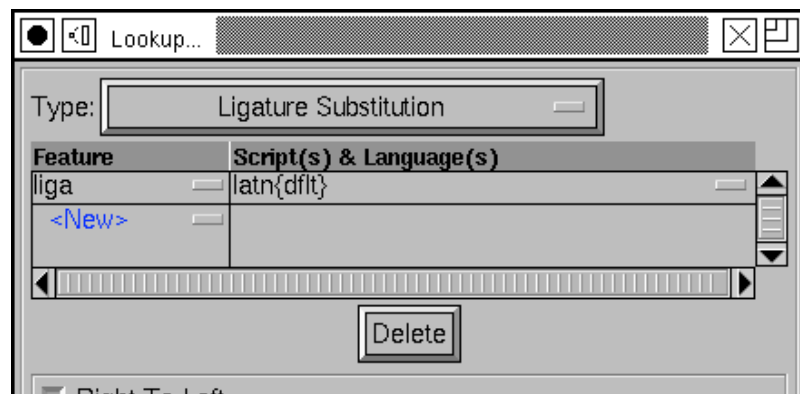
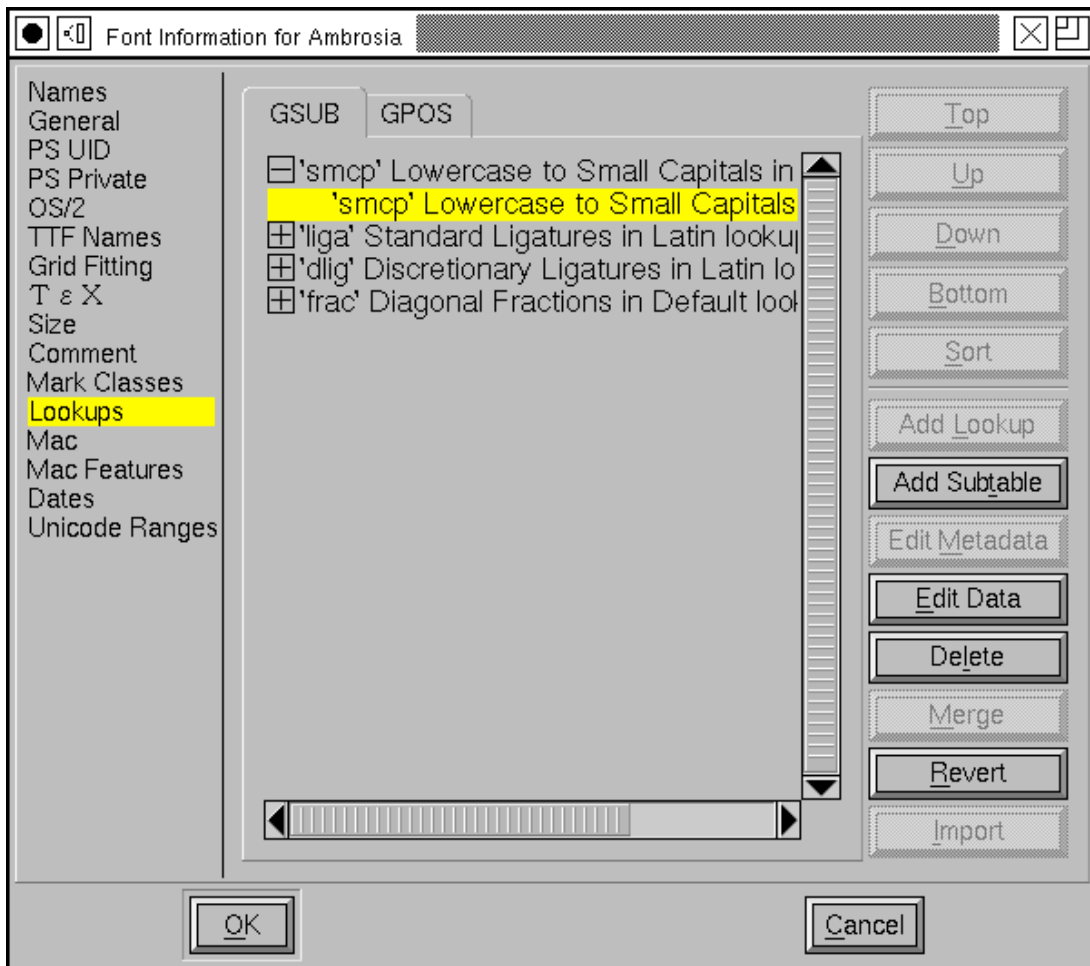
Unfortunately simply creating a ligature glyph is not enough. You must also include information in the font to say that the glyph is a ligature, and to say what components it is built from.

In OpenType this is handled by lookups and features. A lookup is a collection of tables in the font which contain transformation information. A feature is a collection of lookups and is a provides semantic information to the world outside the font about what that set of lookups can be expected to do. So in the example above the lookup would contain the information that "f" + "f" + "i" should turn into "ffi", while the feature would say that this is a standard ligature for the latin script.

So the first time you create a ligature glyph you will need to create a lookup (and a lookup subtable) in which the information for that glyph will reside. Any subsequent ligatures can probably share the same lookup and subtable.

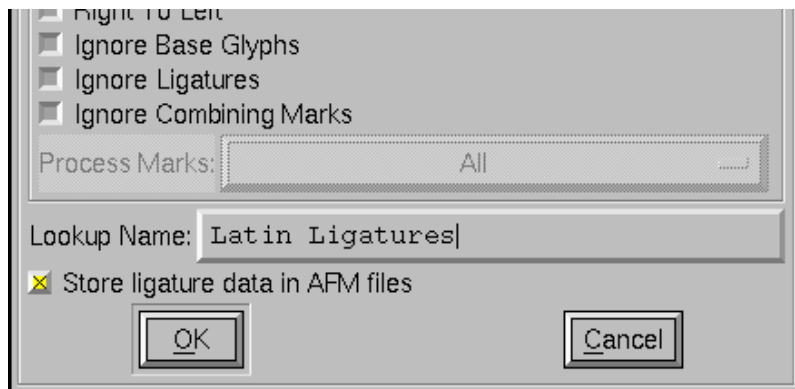
(This may seem like overkill for latin ligatures, and perhaps it is, bt the complexity is needed for more complex writing systems).

You would open the Lookups pane of the [Element->FontInfo](#) command and press the [Add Lookup] button. This will give you a new dialog in which you can fill in the attributes of your new lookup.



You must first choose the lookup type. For ligatures this should be "Ligature Substitution". You may then bind this lookup to a feature, script and language set. The "ffi" ligature is a standard ligature in latin typesetting so it should be bound to the 'liga' tag, and the 'latn' script. (If you click on the little box to the right of "liga" you will get a pulldown list of the so-called "friendly names" for the features. "liga" corresponds to "Standard Ligatures").

The language is a bit tricky. This ligature should probably be active for all languages except Turkish that



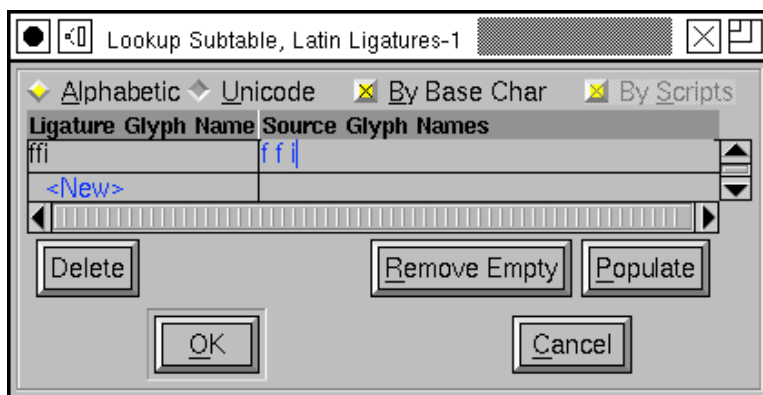
use the latin script (Turkish uses a dotlessi and it is not clear whether the "i" in the "fi" and "ffi" ligatures has a dot over it). So we want to list all languages but Turkish. That's a lot of languages. The convention instead is that if a language isn't mentioned explicitly anywhere in the font then that language will be treated as the "default" language. So to make this feature not be active for Turkish, we'd have to create some other feature which specifically mentioned Turkish in its language list.

Underneath the feature list is a set of flags. In latin ligatures none of these flags need be set. In Arabic one might want to set both "Right to Left" and "Ignore Combining Marks".

Next the lookup must be given a name. This name is for your use and will never be seen in the real font. The name must be distinct from the name of any other lookup however.

Finally you get to decide whether you want the ligatures in this lookup to be stored in afm files.

Once you have created a lookup, you must create a subtable in that lookup. Select the lookup line (in the Lookups pane of Font Info) and press [Add Subtable]. This is a fairly simple dialog... you simply provide a name for the sub-table, and then another dialog will pop up and you will (finally) be able to store your ligature information.

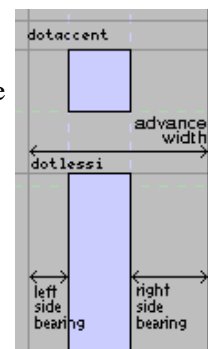


**CAVEAT:** OpenType engines will only apply features which they believe are appropriate for the current script (in Latin scripts, Uniscribe will apply 'liga'). Even worse, some applications may choose not to apply any features ever (Word does not do ligatures in latin -- though this may have changed with the 2007 release?). [Microsoft tries to document](#) what features they apply for which scripts in Uniscribe, but that isn't very helpful since Word and Office have quite different behavior than the default.

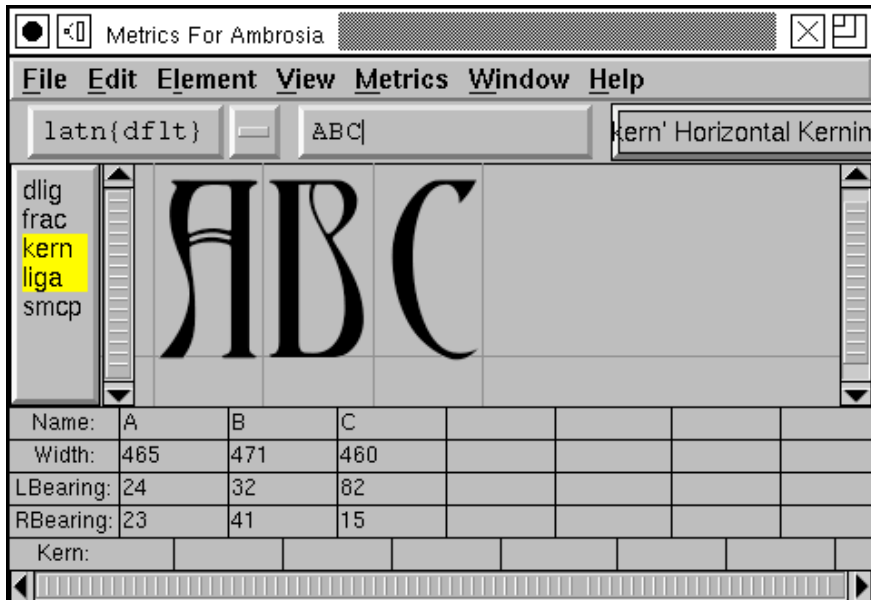
## Examining and controlling metrics

After you have created the shapes of your glyphs you must next figure out the spacing between glyphs. The space between any two glyphs has two components, the space after the first glyph, and the space before the second glyph. In a left to right world these two are called the right side bearing and the left side bearing respectively.

The left side bearing may be changed by the simple expedient of `Edit->Select All` (in the outline view) and then dragging the selected objects to the appropriate place. The right side bearing may be changed by selecting the advance width line and adjusting it appropriately.



However it is generally better not to set the metrics of a single glyph in isolation, you should see the glyph in the context of other glyphs and set it from that perspective. Use the `Window->Open Metrics Window` command.



Any glyphs selected in the fontview (when you invoke the metrics view) will be displayed in the metrics view. You may change which glyphs are displayed by either typing new ones in to the text field at the top of the view, or by dragging a glyph from the fontview.

From here you may adjust any glyph's metrics by typing into the textfields below it, or you may select a glyph (by clicking on its image) and drag it around (to adjust the left side bearing), or drag its width line (to adjust its right side bearing).

If you want to generate a "typewriter" style font (where all glyphs have the same width) execute an `Edit->Select All` from the fontview and then `Metrics->Set Width`. This will set the widths of all glyphs to the same value. After doing that you might also want to execute `Metrics->Center in width` to even out the left and right spacing on each glyph.

If all this manual operation seems too complicated, try [Metrics->Auto Width](#). This will automatically assign widths to glyphs. These widths are not up to professional standards, but they are generally reasonable approximations.

## Vertical Metrics

FontForge provides some support for the vertical metrics needed for CJK fonts. First you must tell FontForge that this font should contain vertical metrics, do this with `Element->Font Info->General->Has Vertical Metrics`. Then in each outline glyph enable VMetrics in the Layers palette.

You should now see a vertical advance line somewhere underneath your glyph. You may drag this line just as you would the horizontal advance (width) line.

## Setting the baseline to baseline spacing of a font.

You might imagine that there would be an easy way to set this seemingly important quantity. Unfortunately there is not.

In a PostScript Type1 font (or bare CFF font)

There is no way to set this value.

At all, ever.

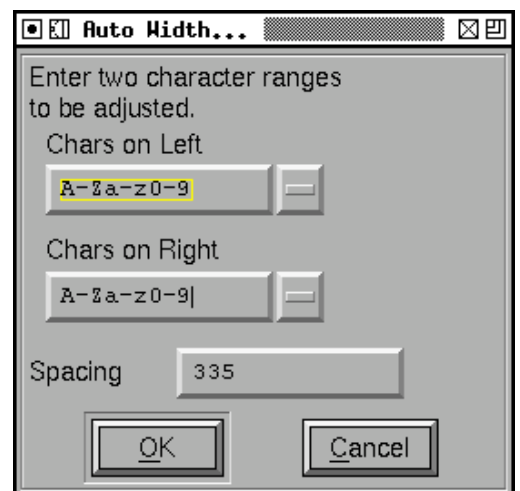
In traditional typography the inter-line spacing is 1em (which in FontForge is the ascent+descent of a font). Some applications will use this. Other applications will use the font's bounding box (summing the maximum ascender height with the minimum descender depth) -- a very bad, but very common approach.

In a TrueType or OpenType font

Unfortunately this depends on the platform

Mac

On a mac the baseline to baseline spacing is determined again by the bounding box values of the font, specified in the 'hhea' table, possibly modified by a linegap (Which you can set in FontForge with [Element->FontInfo-](#)



>OS/2.

#### On Windows

According to the OpenType spec, the baseline to baseline distance is set by the values of Typographic Ascent and Descent of the 'OS/2' table. These can be set with [Element->FontInfo->OS/2](#), but are usually allowed to default to the Ascent and Descent values of FontForge -- they generally sum to 1em and are equivalent to the traditional unleaded default.

Again this may be modified by a linegap field.

Unfortunately Windows programs rarely follow the standard (which I expect doesn't surprise anyone), and generally they will use the font's bounding box as specified in the Win Ascent/Descent fields of the 'OS/2' table.

#### On linux/unix

I doubt there is any standard behavior. Unix apps will probably choose one of the above.

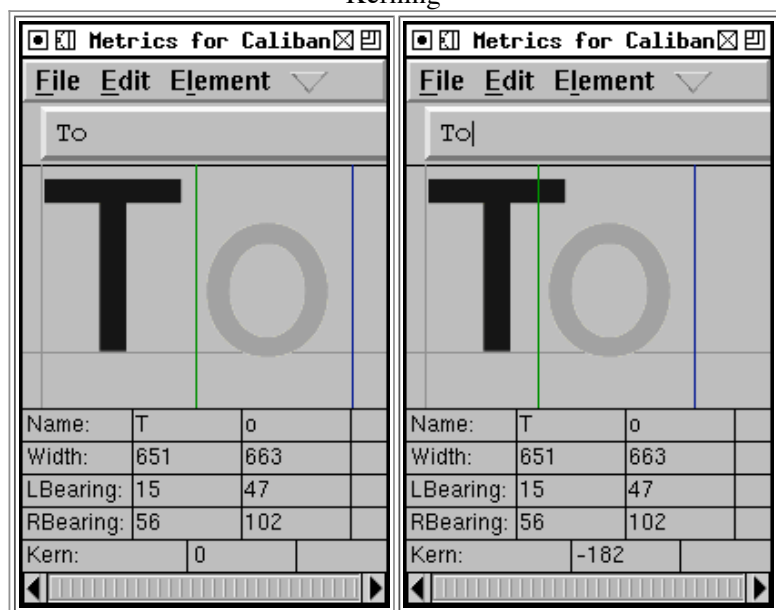
Typographically ept applications will allow users to adjust baseline to baseline spacing, so the default value may not be all that relevant.

## Kerning

If you are careful in setting the left and right side-bearings you can design your font so that the spacing looks nice in almost all cases. But there are always some cases which confound simple solutions.

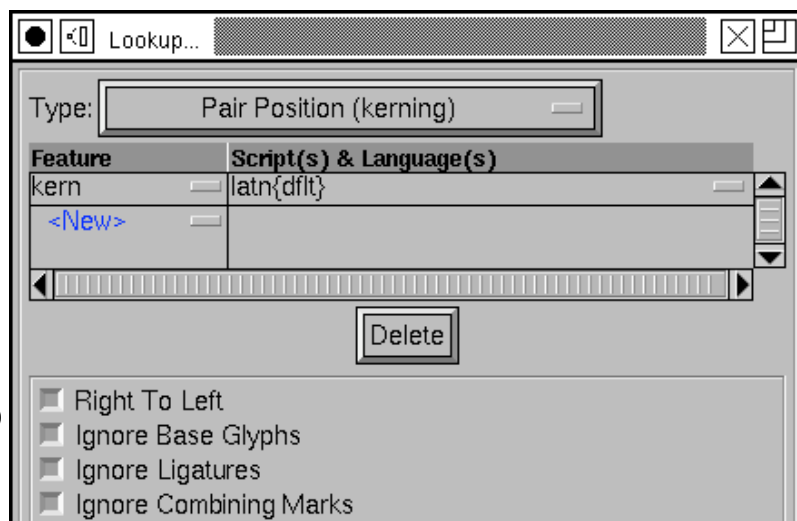
Consider "To" or "Av" here the standard choices are inappropriate. The "o" will look better if it can slide more to the left and snuggle under the top bar of the "T". This is called kerning, and it is used to control inter-glyph spacing on a pair-by-pair basis.

Kerning



In the above example the left image shows the unkerneled text, the right shows the kerned text. To create a kerned pair in the metrics window, simply click on the right glyph of the pair, the line (normally the horizontal advance) between the two should go green (and becomes the kerned advance). Drag this line around until the spacing looks nice.

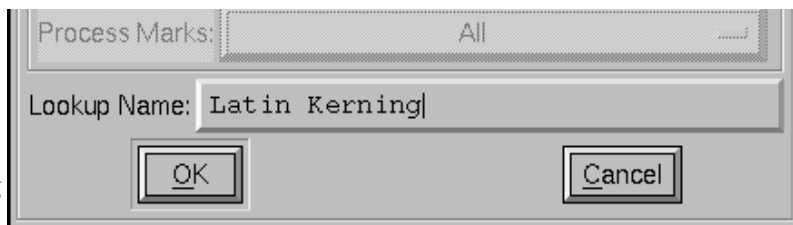
Sadly that statement is a simplification... Before you can create a kerning pair you must create a kerning lookup (see [the section on lookups](#)). Once again you bring up the Element->Font Info->Lookups pane and this time you must select the GPOS (Glyph Positioning) tab at the top of the pane. Once again you press [Add Lookup]. This time the lookup type is "Pairwise Positioning", and the feature is "kern" (or perhaps



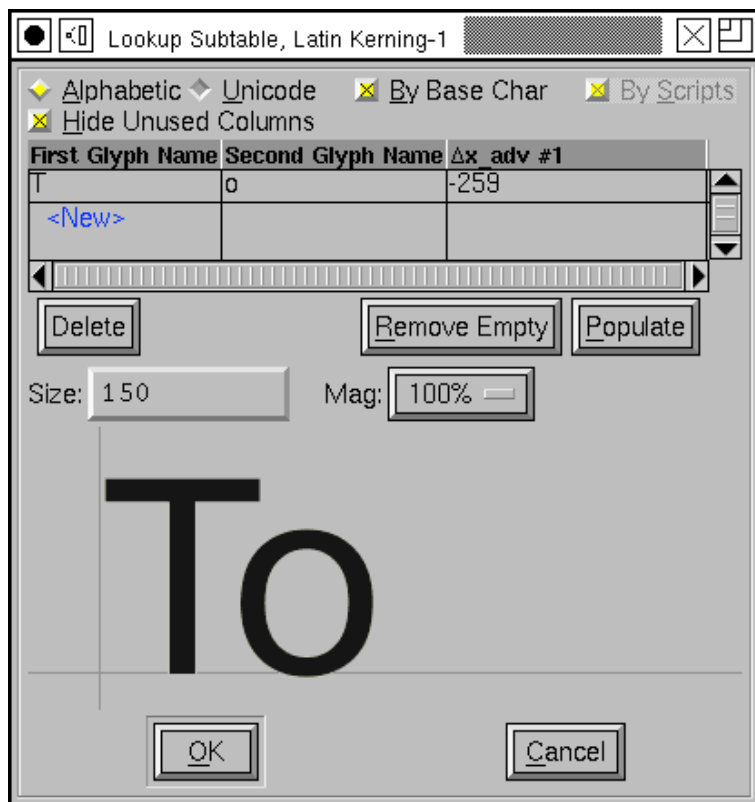


"vkrn" if you want to do vertical kerning).

Once you have created your lookup you again select it and press the [Add Subtable] button (which asks you to name the subtable). Then FontForge will ask you whether you want a subtable of kerning pairs or kerning classes.



If you have many glyphs which have similar kerning features you might wish to create a set of [kerning classes](#) (which might say that A, Â, Ã, Ä, Å and Æ all kern alike). However for this example you want a kerning pair subtable.



Then FontForge will popup a dialog allowing you to set the kerning subtable directly. You may set your kerning pairs from here, though I prefer the metrics view myself because you can see more glyphs and so get a better feel for the "color" of the font.

(Some glyph combinations are better treated by [creating a ligature](#) than by kerning the letters)

## Vertical Kerning

FontForge has equivalent support for vertical kerning. It can read and write vertical kerning information from and to truetype, opentype and svg fonts. It allows you to create vertical kerning classes. The metrics window has a vertical mode in which you can set vertical kerning pairs. Finally it has a command which will copy horizontal kerning information to the vertically rotated glyphs (That is, if the combination "A" "V" is horizontally kerned by -200, then "A.vert" "V.vert" should be vertically kerned by -200.

(Vertical kerning is only available if the font has vertical metrics)

## Glyph Variants

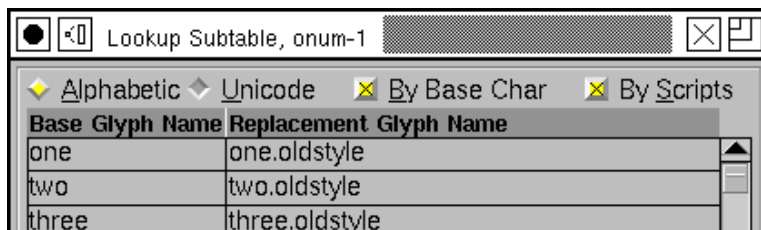
In many scripts glyphs have several variant glyphs. In latin the most obvious example is that every letter has both an upper case and a lower case variant. A more esoteric example would be that in renaissance times the long-s variant (of s) was used initially and medially in a word, while the short-s was only used at the end of a word.

Most Arabic glyphs have four variants (initial, medial, final and isolated).

The digits often have several variants: tabular digits (where all digits have the same advance width so that tables of numbers don't look ragged), proportional digits (where each digit has a width appropriate to its shape) and old-style or lower case digits (123456789) where some digits have descenders and others have ascenders.

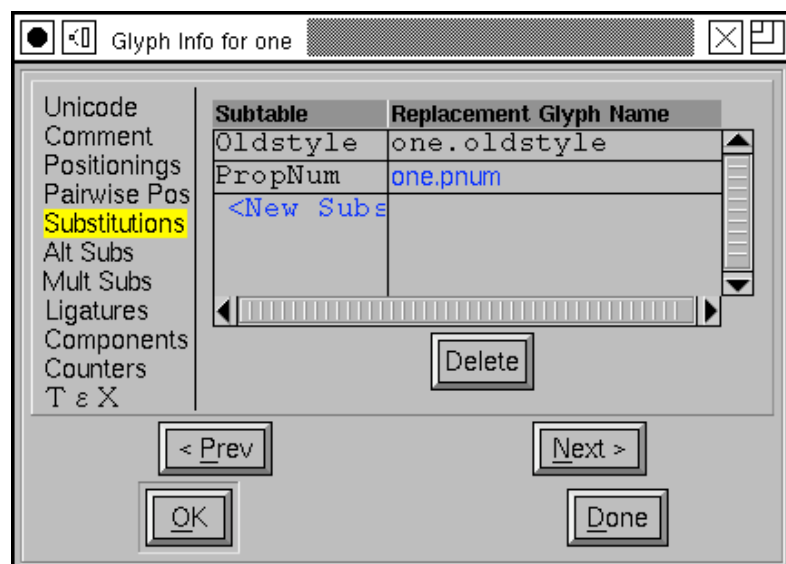
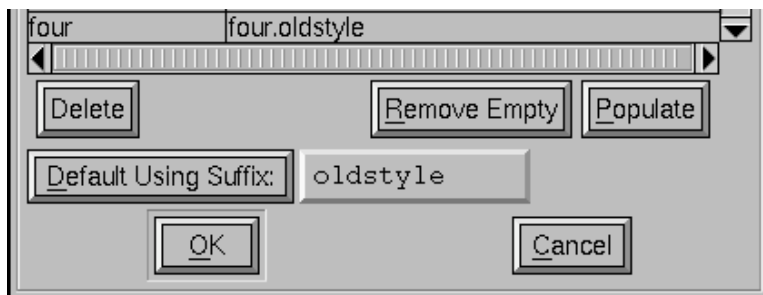
Some of these variants are built into the encodings (the upper and lower case distinction is), but in other cases you should provide extra information in the font so that the word processor can be aware of the variants (Arabic is midway between the two, the various forms are in the encoding, but you need to provide extra information as well).

Let us consider the case of the digits mentioned above. Assume that the glyph called "one" contains the tabular variant of one, the glyph "one.prop" contains the proportional variant and "one.oldstyle" contains the lower-case variant, and so on for all the other digits. Before you do anything else you must create two [lookups](#) and associated subtables (I shan't go into that again. Here the



lookup type is "Single Substitution", and the features are "pnum" for Proportional Numbers and "onum" for Oldstyle Figures. Also the digits aren't in any single script, but are in many, so make this feature apply to multiple scripts (including "DFLT").

When FontForge brings up the dialog to fill in the oldstyle lookup subtable notice that there is a button [Default Using Suffix:] followed by a text field containing a suffix. Set the text field to "oldstyle" and press the button. It will search through all glyphs in all the scripts of the feature and find any "oldstyle" variants of them and populate the table with them.



Sometimes it makes more sense to think of all the substitutions available for a specific glyph (rather than all substitutions in a specific lookup). So instead of filling up the subtable dialog for "Proportional Numbers" let us instead select "one" from the fontview, [Element->Glyph Info](#), select the Substitutions tab and press the <New> button.

(Note: Type0, Type1 and Type3 PostScript fonts have no notation to handle this. You need to be creating an OpenType or TrueType font for these variants to be output).

## Conditional Variants

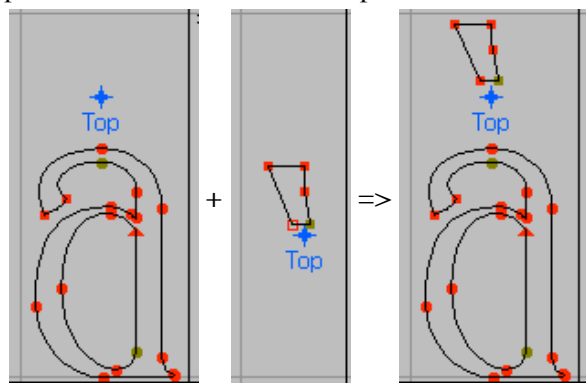
FontForge supports OpenType's Contextual Substitution and Chaining Contextual Substitution sub-tables, and to a lesser extent, Apple's contextual glyph substitution

sub-table. This means that you can insert conditional variants into your font. [The next page](#) will go into this in greater detail.

## Anchoring marks

Some scripts (Arabic, Hebrew) need vowel marks placed around the main text glyphs. Other scripts (some variants of Latin and Greek) have so many possible accent combinations that having preassembled glyphs for all combinations is unwieldy.

In OpenType (which includes Microsoft's TrueType fonts) it is possible to indicate on all base glyphs where marks should attach, and on all marks where the base glyphs should attach. Thus one could put an anchor centered above a lowercase-a indicating that all accents (acute, grave, umlaut, circumflex, tilde, macron, ring, caron, ...) should attach there, and underneath all the accents one could put another anchor so that when the two glyphs are adjacent in the text the word-processor will know where to place the accent so that it rides above the "a".



Not all accents ride centered above the letter (the dot and ogonek ride below the letter), so you may need more than one anchor for different styles of attachment.



Finally some letters can have multiple attachments, unicode U+1EA4, for example, is an A with a circumflex and an acute. Normally the circumflex and the acute will attach at the same point, which would be ugly and confusing. Instead we create a different kind of anchor, a mark to mark anchor, on the circumflex and allow the acute accent to attach to that.

Before one can create an anchor in a glyph one must (of course) create a lookup and subtable. This is another Glyph Positioning lookup (so you enter it in the GPOS pane). Once you have created the subtable you will be presented with another dialog asking for anchor classes, you must create an [anchor class](#) for each type of attachment (thus in the case of A above with two types of attachments (one above and one below) you would create two anchor classes.

Then for each glyph in which an attachment will be made, you should first click at the point where the anchor is to be created and then bring up the [Point->Add Anchor](#) dialog.

You can examine (and correct) how a glyph fits to any others that combine with it by using the [View->Anchor Control...](#) command.

**A warning about mark attachments:** Not all software supports them. And even more confusing software may support them for some scripts and not for others.

## Conditional Features

OpenType and Apple fonts both provide contextual features. These are features which only take place in a given context and are essential for typesetting Indic and Arabic scripts. In OpenType a context is specified by a set of patterns that are tested against the glyph stream of a document. If a pattern matches then any substitutions it defines will be applied. In an Apple font, the context is specified by a state machine -- a mini program which parses and transforms the glyph stream.

Conditional features may involve substitutions, ligatures or kerning (and some more obscure behaviors). First I shall provide an example of a contextual substitution, later of [contextual ligatures](#).

Instead of an Indic or Arabic example, let us take something I'm more familiar with, the problem of typesetting a latin script font where the letters ``b," ``o," ``v" and ``w" join their following letter near the x-height, while all other letters join near the baseline. Thus we need two variants for each glyph, one that joins (on the left) at the baseline (the default variant) and one which joins at the x-height. Let us call this second set of letters the ``high" letters and name them ``a.high," ``b.high" and so forth.



## OpenType Example

### Warning

The following example may not work! The font tables produced by it are all correct, but the designers of OpenType (or its implementors) decided that the latin script does not need complex conditional features and many implementations of OpenType do not support them for latin. This is not even mentioned in the standard, but is hidden away in supplemental information on microsoft's site.

Why do I provide an example which doesn't work? It's the best I can do. If I knew enough about Indic or Arabic typesetting I would provide an example for those scripts. But I don't. The procedures are the same. If you follow them for some other scripts they will work.

On some systems and in some applications 'calt' is supported for latin and this example will work. On other systems/applications the example can be made to work by replacing the 'calt' feature tag (conditional alternatives) with a Required tag (but I gather that is now deprecated).

We divide the set of possible glyphs into three classes: the letters ``bovw", all other letters, and all other glyphs. We need to create two patterns, the first will match a glyph in the ``bovw" class followed by another glyph in the ``bovw" class, while the second will match a glyph in the ``bovw" class followed by any other letter. If either of these matches the second glyph should be transformed into its high variant.

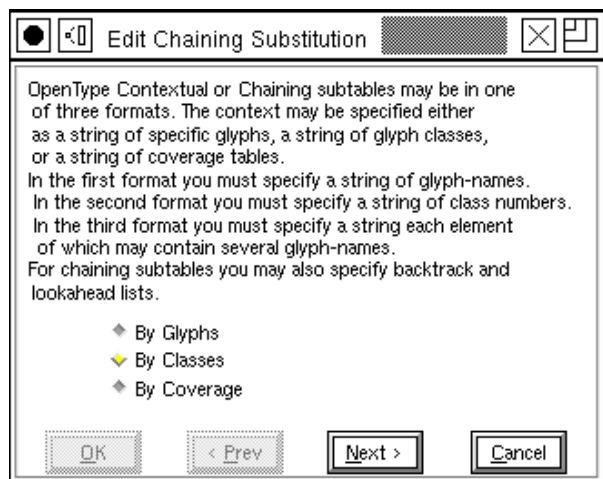
- [bovw] [bovw] => Apply a substitution to second letter

- [bovw] <any other letter> => Apply a substitution to the second letter

(You might wonder why I don't just have a class of all letters and use one rule instead of two? Because in this case all my classes must be disjoint, I mayn't have one glyph appearing in two classes).

The first thing we must do is create a simple substitution mapping each low letter to its high variant. This is a "Simple Substitution" lookup, but it will not be attached to any feature, instead it will be invoked by a contextual lookup. Let us call this lookup "high". We must (of course) create a subtable to go with our lookup, and we can use the [Default with Suffix:] button to fill it up with the high variants.

The tricky part is defining the context. This is done by defining yet another lookup, a contextual chaining lookup which should be associated with a 'calt' feature. And of course we want an associated subtable). This will pop up a series of dialogs to edit a contextual subtable



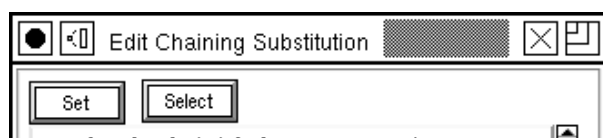
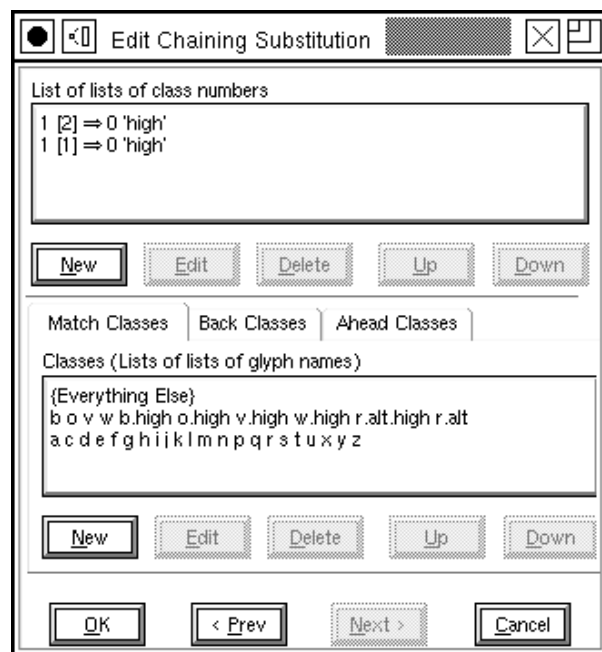
The first dialog allows you to specify the overall format of the substitution. We want a class based system -- we've already mentioned what the glyph classes will be.

The next dialog finally shows something interesting. At the top are a series of patterns to match and substitutions that will be applied if the string matches. Underneath that are the glyph classes that this substitution uses. A contextual chaining dialog divides the glyph stream into three categories: those glyphs before the current glyph (these are called backtracking glyphs), the current glyph itself (you may specify more than one), and this (these) glyphs may have simple substitutions applied to them, and finally glyphs after the current glyph (these are called lookahead glyphs).

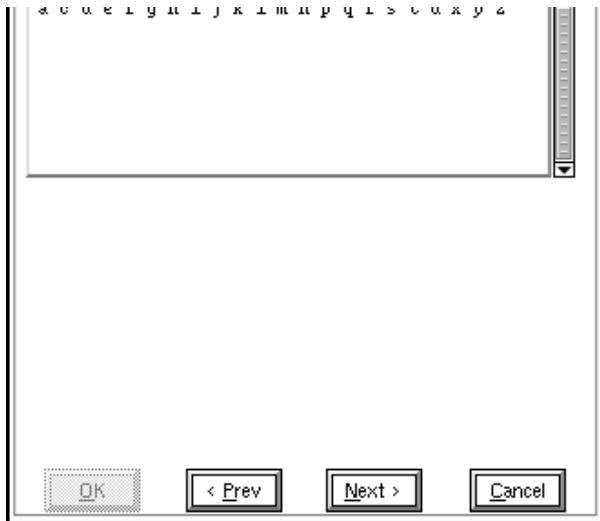
Each category of glyphs may divide glyphs into a different set of classes, but in this example we use the same classes for all categories (this makes it easier to convert the substitution to Apple's format). The first line (in the "List of lists" field) should be read thus: If a backtracking glyph (the glyph before the current one) in class 1 is followed by the current glyph in class 2, then location 0 --the only location -- in the match string (that is the current glyph) should have simple substitution 'high' applied to it.

If you look at the glyph class definitions you will see that class 1 includes those glyphs which must be followed by a high variant, so this seems reasonable.

The second line is similar except that it matches glyphs in class 1. Looking at the class definitions we see that classes 1 & 2 include all the letters, so these two lines mean that if any letter follows one of "bovw" then that letter should be converted to its 'high' variant.



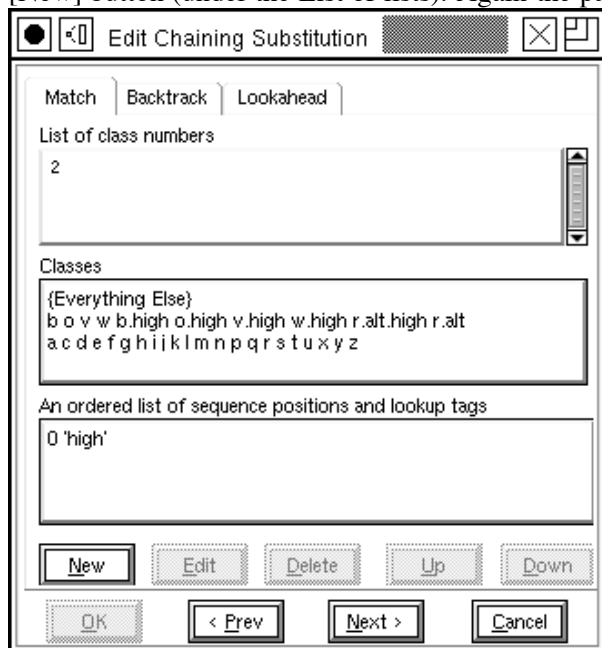
To edit a glyph class simply double click on it. To create a new one press the [New] button (under the class list). This produces another dialog showing all the names of all the glyphs in the current class. Pressing the [Select] button will set the selection in the font window to



match the glyphs in the class, while the [Set] button will do the reverse and set the class to the selection in the font window. These provide a short cut to typing in a lot of glyph names.

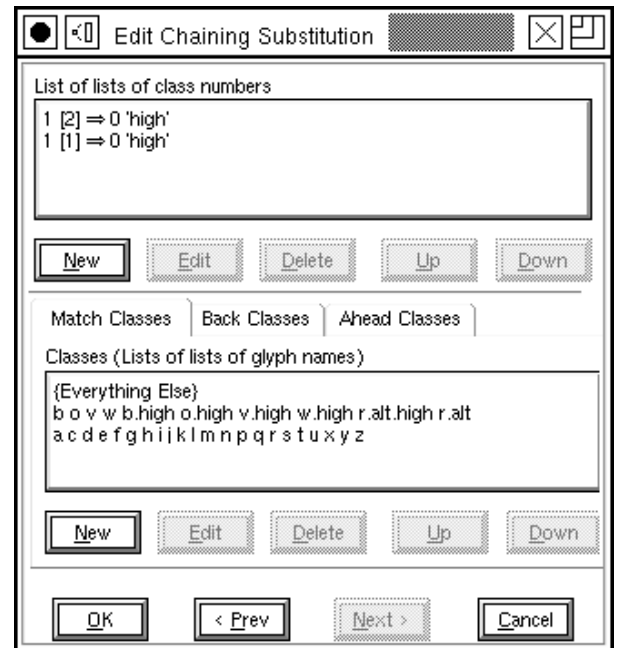
Pressing the [Next] button defines the class and returns to the overview dialog.

To edit a pattern double click on it, or to create a new one press the [New] button (under the List of lists).



into three categories, those glyphs before the current one, the current one itself, and any glyphs after the current one. You choose which category of the pattern you are editing with the tabs at the top of the dialog.

Underneath these is the subset of the pattern that falls within the current category, the classes defined for this category, and finally the substitutions for the current glyph(s). Clicking on one of the classes will add the class number to the pattern.



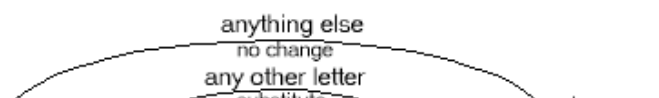
To edit a substitution double click on it, or to create a new one press the [New] button (under ``An ordered list...''). The sequence number specifies which glyph among the current glyphs should be modified, and the tag specifies a four character substitution name

**A warning about contextual behavior:** Not all software supports them. And even more confusing software may support them for some scripts and not for others.

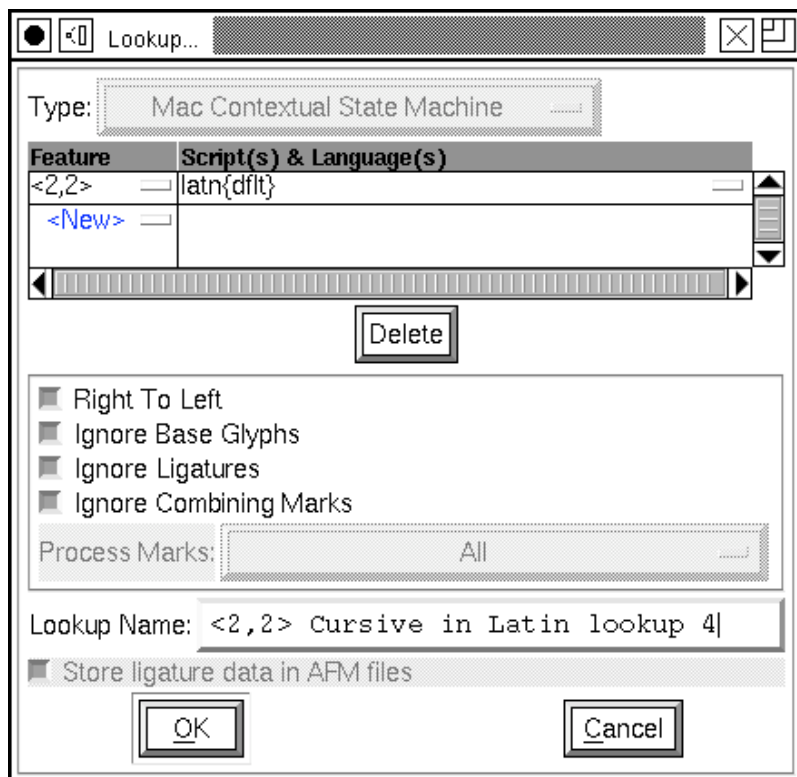
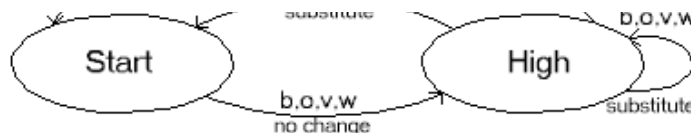
## Apple advanced typography

Apple specifies a context with a finite state machine, which is essentially a tiny program that looks at the glyph stream and decides what substitutions to apply. Each state machine has a set of glyph class definitions (just as in the OpenType example), and a set of states. The process begins in state 0 at the start of the glyph stream. The computer determines what class the current glyph is in and then looks at the current state to see how it will behave when given input from that class. The behavior includes the ability to change to a different state, advancing the input to the next glyph, applying a substitution to either the current glyph or a previous one (the ``marked" glyph).

Using the same example of a latin script font... We again need a simple substitution to convert each letter into its high alternate. The process is the same as it was for OpenType, and indeed we

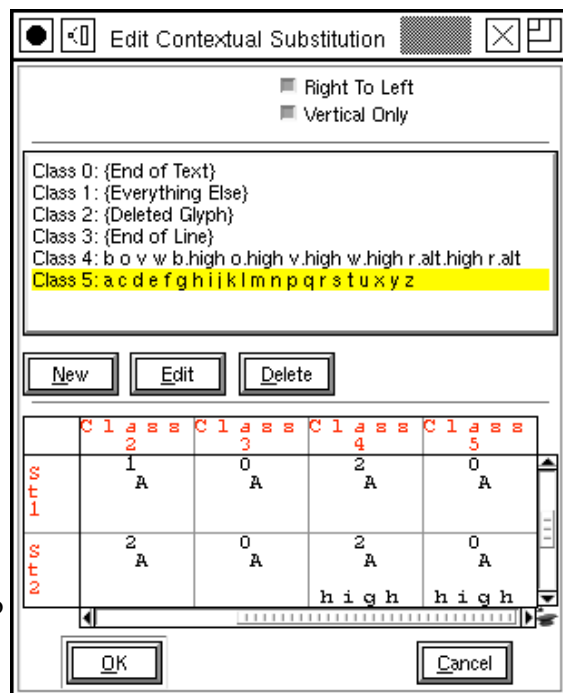


can use the same substitution. Again we divide the glyphs into three classes (Apple gives us some extra classes whether we want them or no, but conceptually we use the same three classes as in the OpenType example). We want a state machine with two states (again Apple gives us an extra state for free, but we shall ignore that), one is the start state (the base state -- where nothing changes), and the other is the state where we've just read a glyph from the ``bovw'' class.

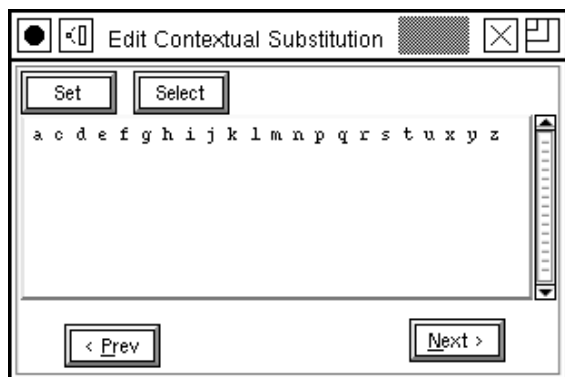


Apple Advanced Typography does not quite fit into the OpenType concepts of lookups and features, but it is close enough that I sort of force it to. So once again we create a GSUB lookup. This time the lookup type is "Mac Contextual State Machine", and the feature is actually a mac feature/setting, two numbers. When we create a new subtable of this type we get a state machine dialog, as shown below.

At the top of the dialog we see a set of class



definitions, and at the bottom is a representation of the state machine itself.

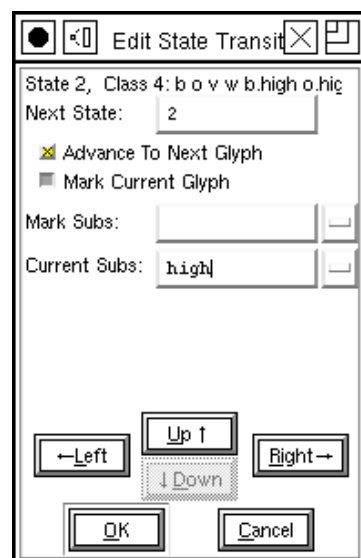


Double clicking on a class brings up a dialog similar to that used in OpenType

``bovw'' glyph), and it has received a glyph in class 4 (which is another ``bovw'' glyph). In this case the next state will be state 2 again (we will have just read a new ``bovw'' glyph), read another glyph and apply the ``high'' substitution to the current glyph.

At the bottom of the dialog are a series of buttons that allow you to navigate through the transitions of the state machine.

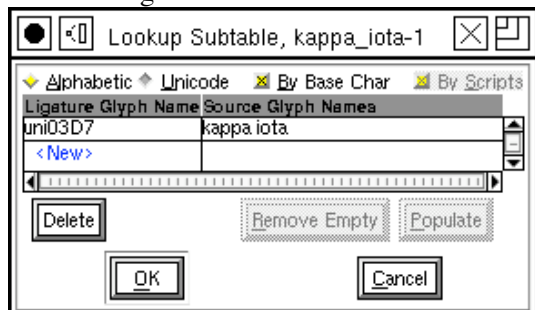
Pressing [OK] many times will extract you from this chain of dialogs and add a new state machine to your font.



## OpenType, Greek ligatures

Greek has a character (U+03D7) which is equivalent to the Latin ampersand. Just as the ampersand is (originally) a ligature of "E" and "t", so U+03D7 is a ligature of "kappa" and "iota". However this ligature should only be used if "kappa" and "iota" make up a word unto themselves, it should not be used for more normal occurrences of the two within a longer word.

So the first thing to do is create the ligature itself. Add the glyph for U+03D7, and then create a ligature lookup and subtable (with [Element->Font Info->Lookups](#)) to bind U+03D7 to be a ligature of "kappa" and "iota". This lookup will never be used directly -- only under the control of another, a conditional feature -- so we don't give it a feature tag.



Next the conditional bit.

I'm going to use the notation <letters> to represent a class consisting of all greek letters.

1. <letters> kappa iota => no substitution
2. kappa iota <letters> => no substitution
3. kappa iota => apply the ligature "WORD"

(Now as I read the standard all these rules could be put into one subtable, and the font validation tools I have agree with me -- but the layout engines do not. The layout engines seem to insist that each rule live in its own subtable. This is inconvenient (the classes must be defined in each subtable) but it seems to work.)

These rules will be executed in order, and the first one that matches the input text will be the (one and only) rule applied. Consider these three strings,  $\alpha\kappa\iota$ ,  $\kappa\iota\theta$ ,  $\alpha\kappa\iota\theta$  all contain kappa and iota but each contains more letters around them, so none should be replaced by the ligature.

- The first string,  $\alpha\kappa\iota$ , will match the first rule above (it contains letters before the kappa iota sequence) and no substitution will be done. It also matches the third rule, but we never get that far.
- The second string,  $\kappa\iota\theta$ , will match the second rule above (it contains letters after the sequence) and again no substitution will be done. It would match the third rule, but we stop with the first match.
- The third string,  $\alpha\kappa\iota\theta$ , matches all the rules, but since the search stops at the first match, only the first rule will be applied, and no substitution will be done.
- The string,  $\kappa\iota$ , matches neither of the first two rules but does match the last, so here the ligature will be formed.

You might wonder why I don't just have one rule

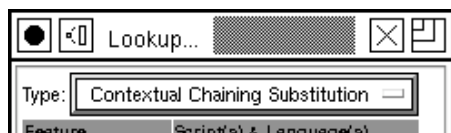
1. <any non-letter> kappa iota <any non-letter> => apply our ligature

It seems much simpler.

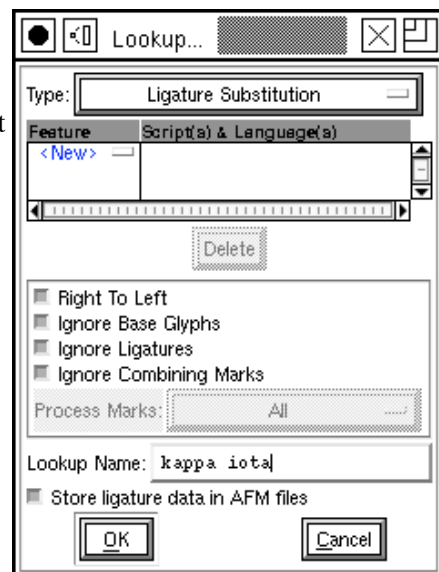
Well there's one main reason:

- This does not work if the kappa is at the beginning of the input stream (it will not be preceded by any glyphs, but might still need replacing), or iota at the end.

Now how do we convert these rules into a contextual lookup?



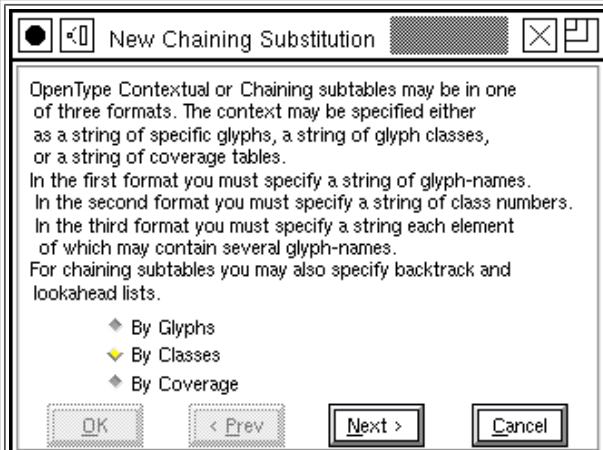
We use [Element->Font Info->Lookups->Add Lookup](#) to create a new contextual chaining lookup. This is the top level lookup and should be bound to a feature tag in the Greek script.



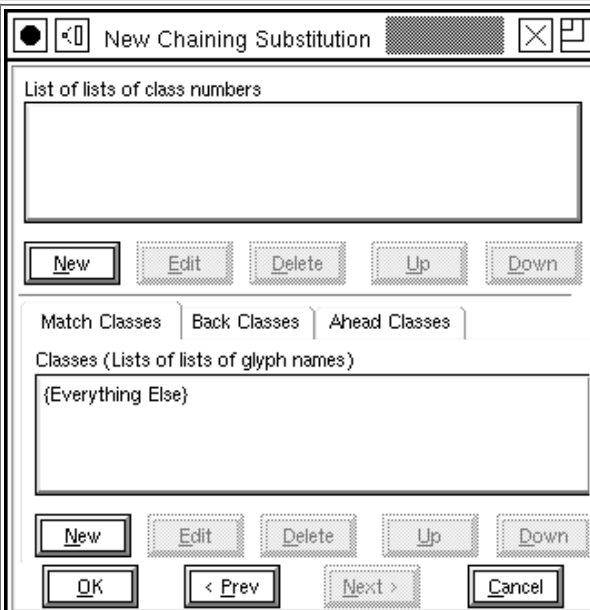




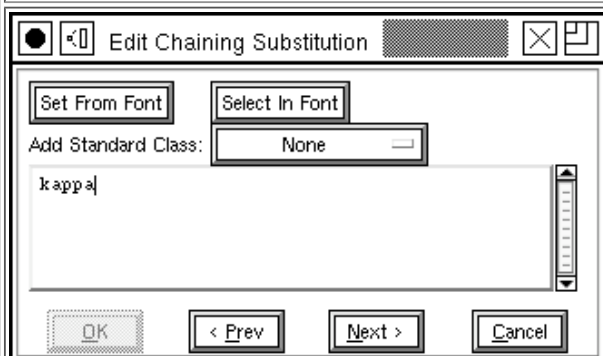
We have three rules, each rule lives in its own subtable, so we will create three subtables, one for each. The order in which these subtables in the Lookups pane is important because that is the order in which the rules they contain will be executed. We must insure that that final rule which actually invokes the ligature is the last one executed (and the last one in the list).



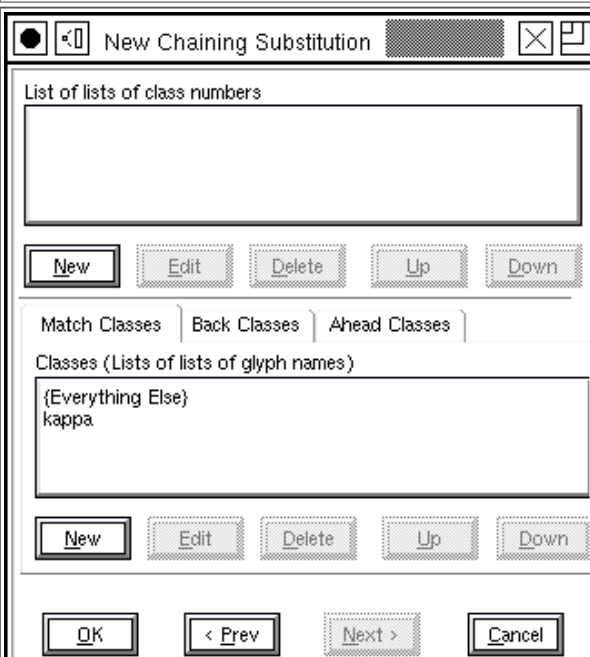
Since we are planning on using the class of all greek letters we will want to use a class format for this feature. Then we press the [Next>] button.



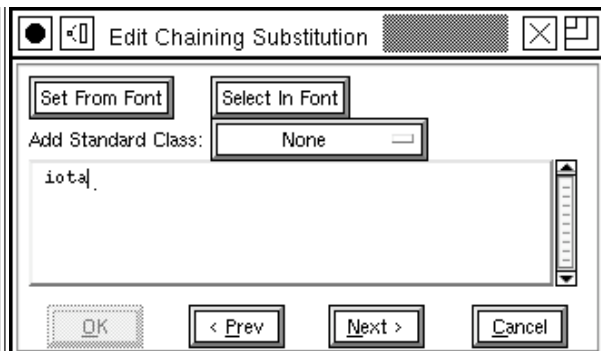
The main match will be on the letters kappa and iota in all three rules, so we need one class for each of them. So in the Match Classes area we press the [New] button...



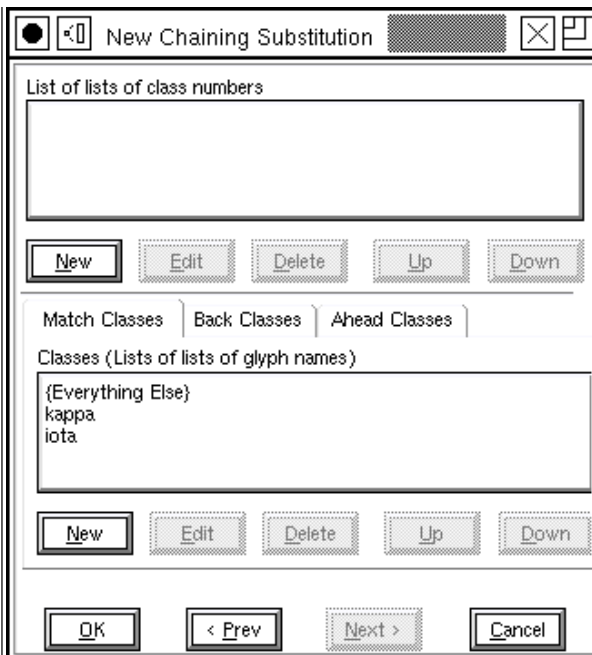
And type in the word "kappa" and press [Next>]



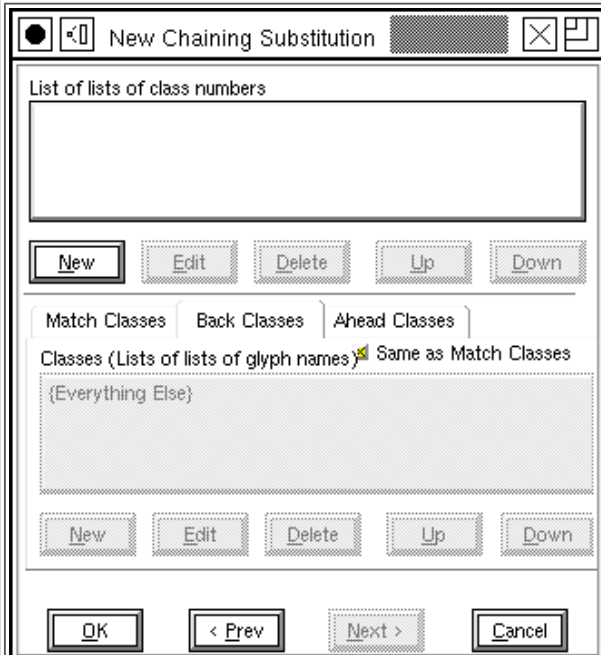
Now we have a class containing the single glyph "kappa". We want to do the same thing for "iota" so we press [New] again.



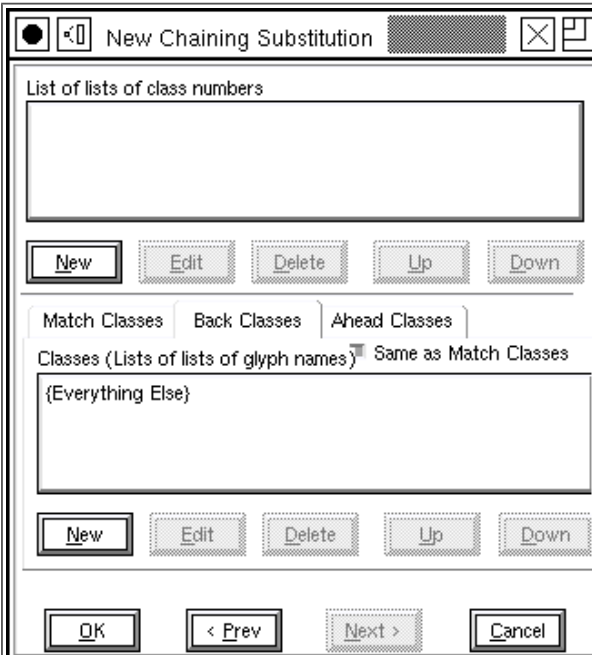
Again type in "iota" and press [Next>]



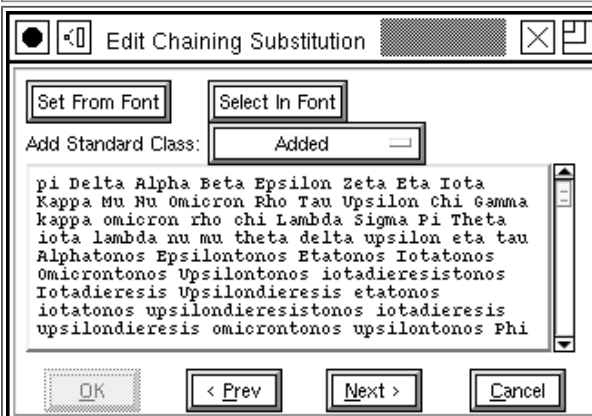
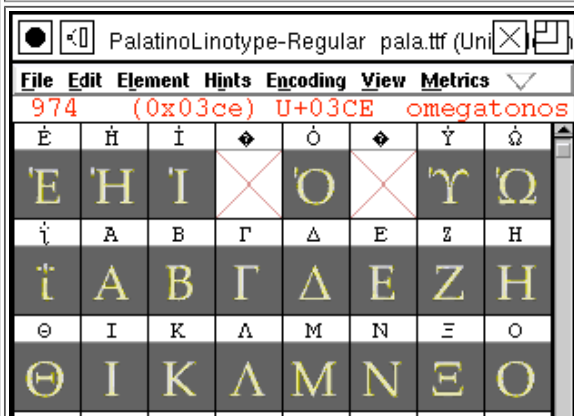
Now we have all the classes we need here. We still need to create classes for the lookahead and backtrack. We only need one class for these groups and that class will consist of all greek letters.



The check box [\*] Same as Match Classes is set, but we don't want that, we want our own classes here. So uncheck it.



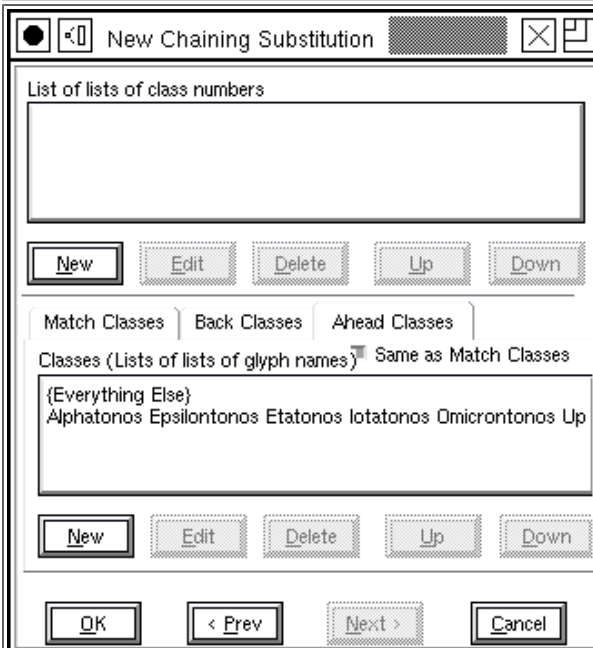
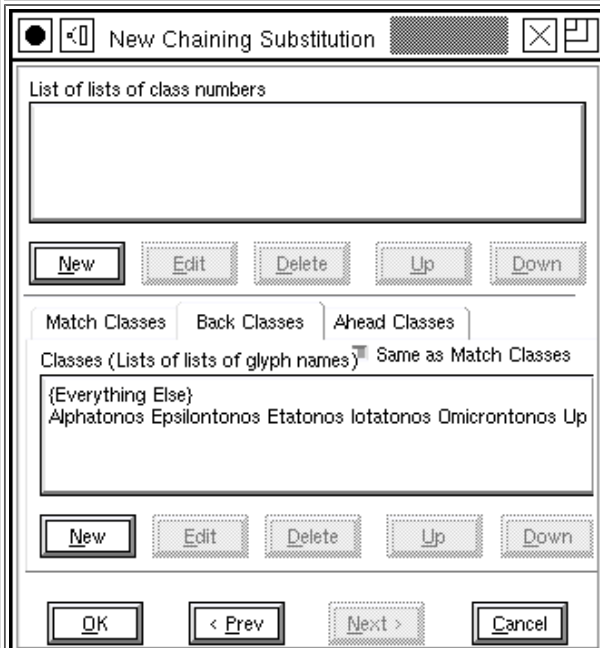
Now the buttons become active and we can create a new class by pressing [New]





Now you could go back to the font view and select all of the greek letters, and then press the [Set From Font] button in the class dialog.

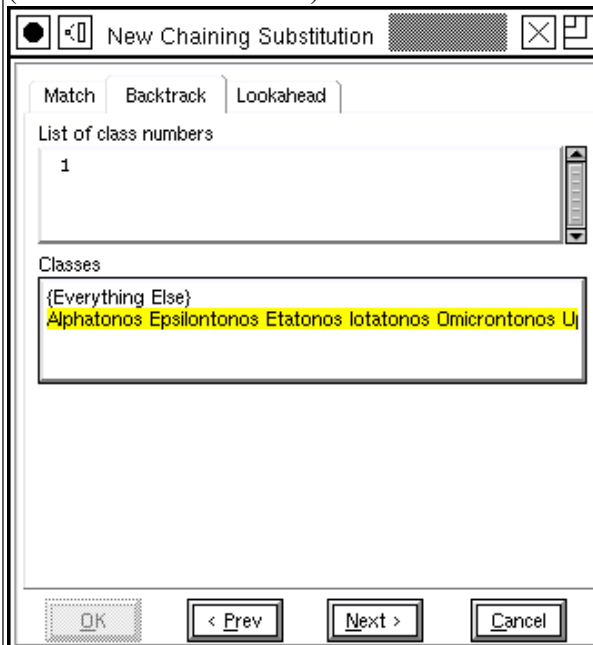
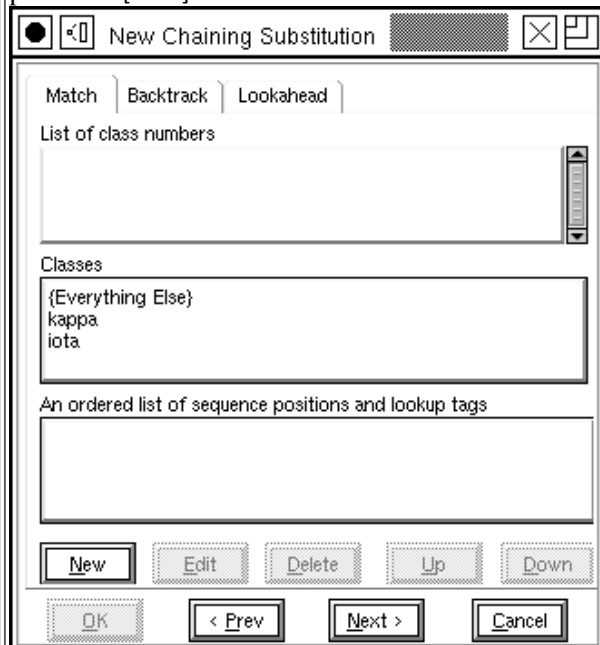
But in this case the class we are interested in (all the greek letters) is built it, and you can select it from the Standard Class pulldown list (Letters in script(s)) Then press [Next>].



Then go through the same process for the look ahead classes (adding one class which consists of all the greek letters).

Now we have all our classes defined and are finally ready to create the patterns for our rules. So underneath "List of lists of class numbers" press the [New] button.

The first rule begins with all the greek letters in the backtrack area, so click on the "Backtrack" tab, and then press on the class consisting of all the greek letters. This puts the class number into the pattern area (the List of class numbers)



In the match area we want to match kappa and then iota, so click on the Match tab, and then on the entries for "kappa" and "iota".

We are done with the first rule. It says:

- The previous character should match class 1 of the backtrack classes (and that class contains all greek letters, which is what we want)
- The current character should match class 1 of the match classes (and that class contains "kappa")



List of lists of class numbers

1 2

Classes

{Everything Else}  
kappa  
iota

An ordered list of sequence positions and lookup tags

New Edit Delete Up Down

OK < Prev Next > Cancel

This rule has no substitutions, so leave the bottom area blank and press [Next>].

- The next character should match class 2 of the match classes (which is iota)
- And if the match is successful, do absolutely nothing.

New Chaining Substitution

List of lists of class numbers

1 [1 2] =>

New Edit Delete Up Down

Match Classes Back Classes Ahead Classes

Classes (Lists of lists of glyph names)

{Everything Else}  
kappa  
iota

New Edit Delete Up Down

OK < Prev Next > Cancel

We've got two more rules though, so press [OK] and then [Add subtable]. Then go through the process of adding all the classes, and then add the match string for this rule.

We are done with the second rule. It says:

- The current character should match class 1 of the match classes (and that class contains "kappa")
- The next character should match class 2 of the match classes (which is iota)
- The character after that should match class 1 of the lookahead classes (and that class contains all the greek letters)
- And if the match is successful, do absolutely nothing.

New Chaining Substitution

List of lists of class numbers

[1 2] 1 =>

New Edit Delete Up Down

Match Classes Back Classes Ahead Classes

Classes (Lists of lists of glyph names)

{Everything Else}  
kappa  
iota

New Edit Delete Up Down

OK < Prev Next > Cancel

New Chaining Substitution

Match Backtrack Lookahead

List of class numbers

1 2

Classes

{Everything Else}  
kappa  
iota

An ordered list of sequence positions and lookup tags

New Edit Delete Up Down

OK < Prev Next > Cancel

This rule does have substitutions -- we want to take the two characters and convert them into a ligature. So Press [New] under the sequence position list, we want to start at the first character (sequence position 0) and apply the ligature we called "WORD":

Sequence/Lookup

Sequence Number:

0

Lookup:

kappa\_iota

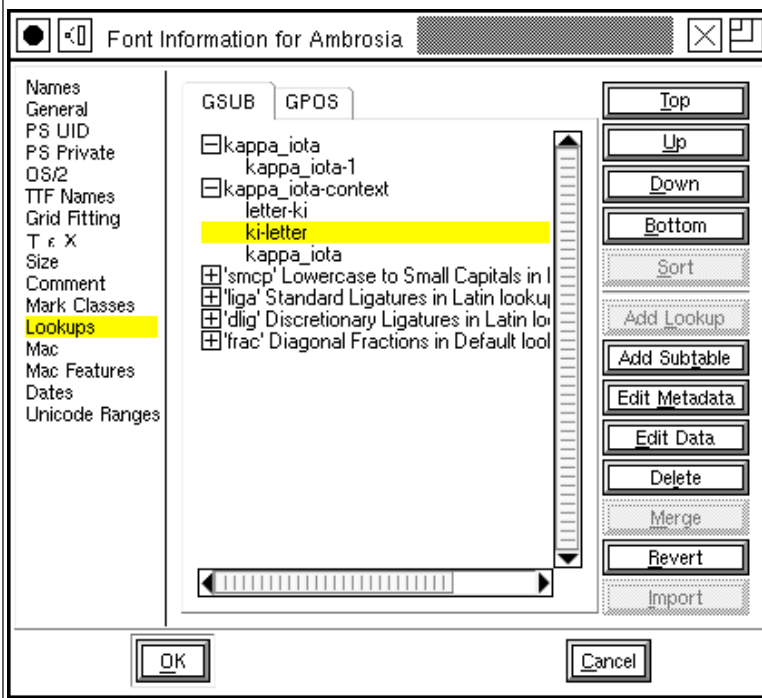
OK Cancel

Press [OK] and [Add Subtable] for the final rule.

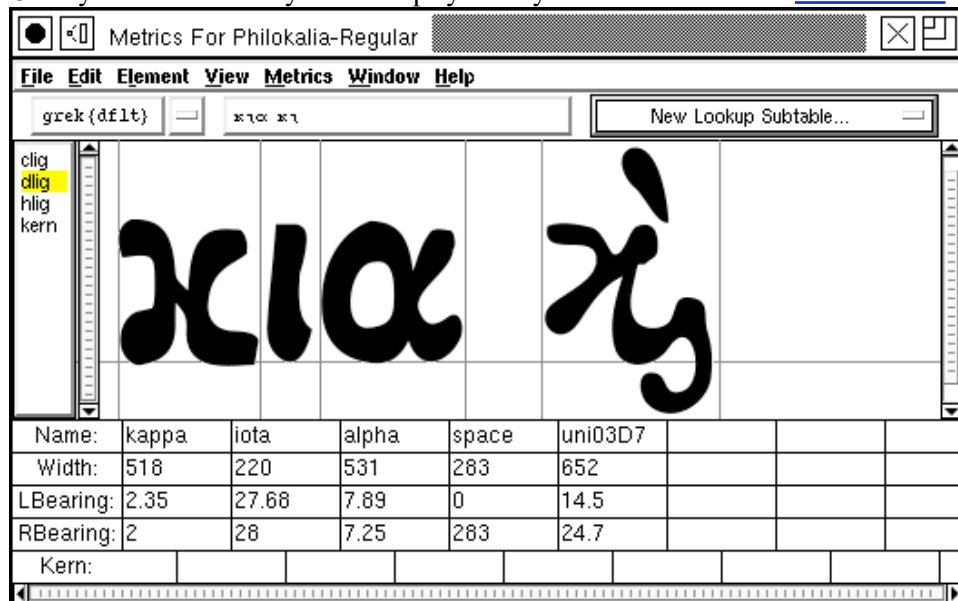
So if anything doesn't match the first two rules, and does contain a kappa followed by an iota, it must be a two letter stand-alone greek word. And we want to apply our ligature to it.



Now we are done. Press a series of [OK]s until all the dialogs have been accepted.



Once you have created your lookups you may test the result in the [metrics view](#).



(This example was provided by Apostolos Syropoulos)

## Checking a font

After you have finished making all the glyphs in your font you should check it for inconsistencies. FontForge has a command, [Element->Find Problems](#) which is designed to find many common problems.

Simply select all the glyphs in the font and then bring up the Find Problems dialog. Be warned though: Not everything it reports as a problem is a real problem, some may be an element of the font's design that FontForge does not expect.

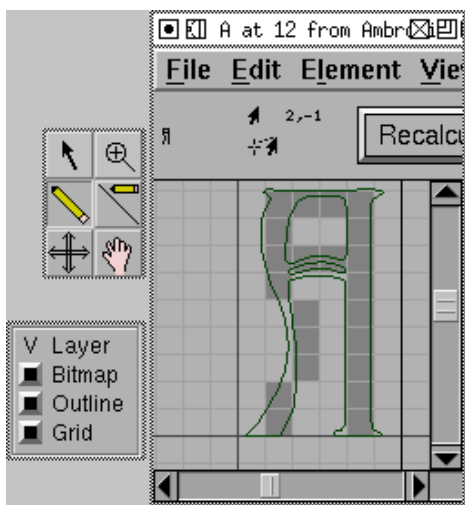
The dialog can search for problems like:

- Stems which are close to but not exactly some standard value.
- Points which are close to but not exactly some standard height
- Paths which are almost but not quite vertical or horizontal
- Control points which are in unlikely places
- Points which are almost but not quite on a hint
- ...

I find it best just to check for a similar problems at a time, otherwise switching between different kinds of problems can be distracting.

## Bitmaps

At this point you might want some bitmaps to go with the outline font (this is not compulsory). Go to **E**lement->Bitmap Strides Available and select the pixel sizes you want bitmaps in (Note, that on X and MS windows pixel sizes often do not correspond exactly to point sizes. You can then use the bitmap editor ([Window->Open Bitmap](#)) to clean up the bitmaps, or you can generate your bitmap fonts and then [use someone else's bitmap editor to clean them up](#).



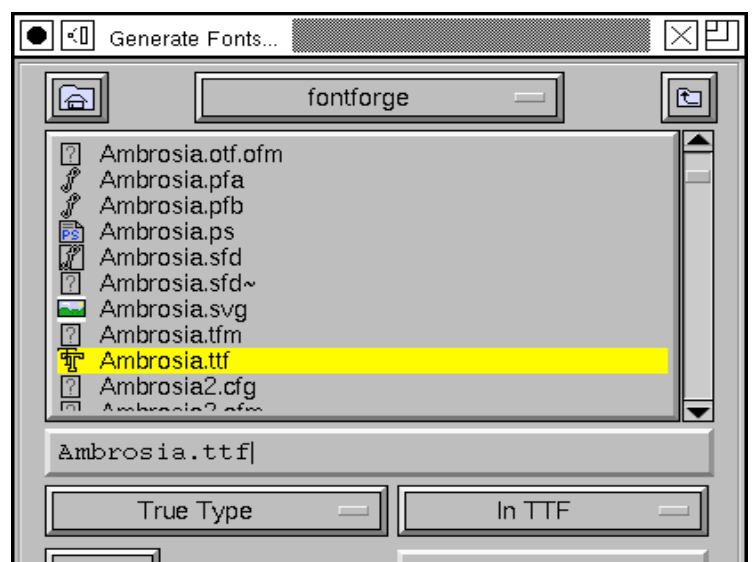
Bitmaps are discussed in more detail in the [next section](#).

## Generating a font

If you save your file it will be saved in a format that only FontForge understands (as far as I know anyway). This is not very helpful if you want to use the font.

Instead you must use [File->Generate](#) to convert your font into one of the standard font formats. FontForge presents what looks like a vast array of font formats, but in reality there are just several variants on a few basic font formats: PostScript Type 1, TrueType, OpenType (and for CJK fonts, also CID-keyed fonts).

You also have a choice of bitmap formats. FontForge supports bdf (used by X), mac NFNT (used by the Mac), Windows FNT (used by Windows 2.0 I think) and storing bitmaps inside true (or open) type wrappers.





## Font Families

After you have generated a font, you probably want to generate a sequence of similar fonts. In Latin, Greek and Cyrillic fonts italic (or oblique), bold, condensed, expanded styles are fairly common.

Fonts with different styles in the same family should share the same Family Name (in the [Element->Font Info->Names](#) dialog). The Font Name should be the Family Name with the style name(s) appended to the end, often preceded by a hyphen. So in the font family "Helvetica" all fonts should have the Family Name set to "Helvetica". The plain style could be called simply "Helvetica" or "Helvetica-Regular", the bold style "Helvetica-Bold", the oblique (Helvetica doesn't have a true italic) "Helvetica-Oblique", etc.

FontForge has a command (which doesn't work well yet, but I hope to improve eventually) [Element->MetaFont](#) which is designed to help you create a bold (condensed, expanded, etc.) style from a plain one.

The [Element->Transform->Transform->Skew](#) command can turn a plain font into an Oblique one. Creating a true italic font is generally a bit more complex, the shape of the "a" changes dramatically to "a", the "f" gains a descender as "f", the serifs on "ilm" etc. become rounded as "ilm" and there will probably be other subtle differences. Also, after having skewed a font you should [Element->Add Extrema](#).

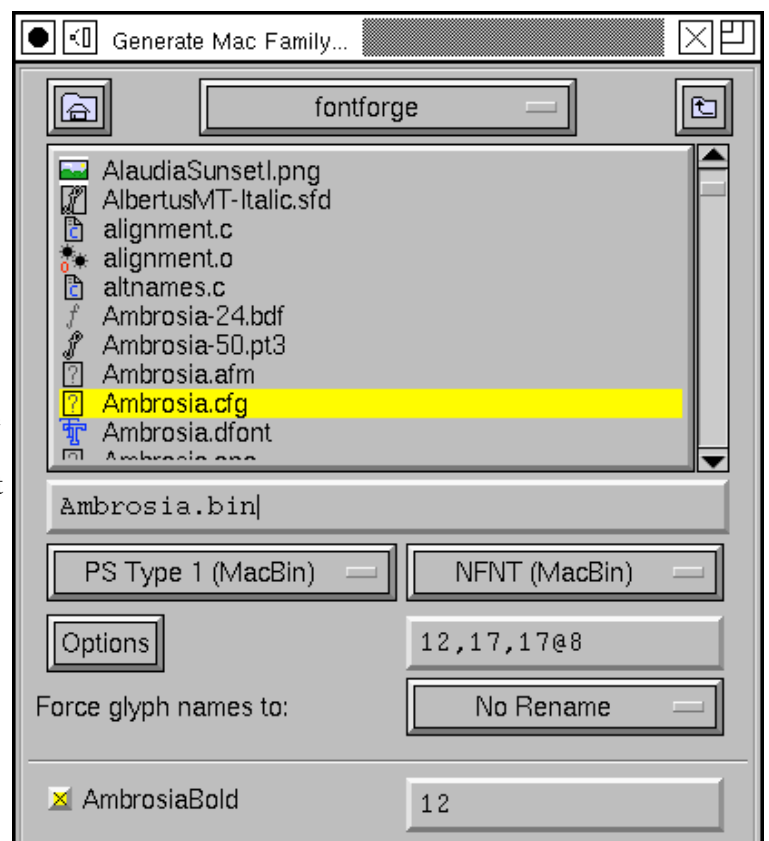
If you already have a "Bold" and a "Plain" style of a font (and each glyph has the same number of points in the same order), you can use the [Element->Interpolate Font](#) command to generate a "DemiBold" style.

TrueType fonts (and Windows) support a very fine gradation of stem thicknesses (the Mac really only understands Plain and Bold). If you go to [Element->Font Info->OS/2](#) you can set the weight to be any value between 0 and 999 (with plain generally being at 400 or 500, and Bold at 700). TrueType fonts also support a range of widths (while the Mac only supports condensed, plain and expanded).

On Windows machines, as long as you get the names right, the system should be able to figure out what fonts go into what families. But on the Mac the situation is (or was, it is changing and I don't understand all the new extensions yet) more complicated. The Mac supports a limited range of styles (plain, italic, bold, outline, condensed, expanded and combinations of these) anything outside these must go into a separate family. Then a special table needs to be constructed (called the FOND) which holds pointers to the various fonts in the family. If you open all the fonts you want to be in a given family (and if they have been given the proper names) and then from the plain font select [File->Generate Family](#). This will list all the fonts that FontForge thinks belong to the same family as the current font and will allow you to generate a FOND structure as well as font files for all family members (sometimes all the fonts live in one file, sometimes they don't, it depends on the font format chosen).

## Final Summary

So you have made a new font. But it does you no good just sitting on your disk, you must install it on your machine. On some systems this is as simple as just dragging the new font into your system Fonts folder, but on other systems



there is a fair amount work involved still. See the [Installing fonts FAQ](#).

For a tutorial about [FontForge's scripting mechanism](#) click [here](#).

