Font extensions

Hans Hagen

1. Introduction

One of the benefits of using TEX is that you can add your own features and try to optimize the look and feel. Of course this can also go wrong and output can look pretty awful when you don't know what you're doing, but on the average it works out well. In many aspects the move to an UNICODE data path and OPENTYPE fonts is a good one and solves a lot of problems with traditional TEX engines and helps us to avoid complex and ugly hacks. But, if you look into the source code of ConTEXt you will notice that there's still quite some complex coding needed. This is because we want to control mechanisms, even if it's only for dealing with some border cases. It's also the reason why LuaTEX is what it is: an extensible engine, building on tradition.
As always with TEX, fonts are an area where many tuning happens and this is also true in ConTEXt. In this chapter some of the extensions will be discussed. Some extensions run on top of the (rather generic) feature mechanism and some are using dedicated code.

2. Italics

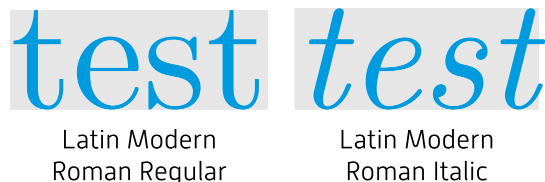
Although OPENTYPE fonts are more rich in features than traditional TEX and TYPE1 fonts, one important feature is missing: italic correction. This might sound strange but you need to keep in mind that in practice it's a feature that needs to be applied manually.

```
test {\it test\/} test
```

It is possible to automate this mechanism and this is what the `\em` command does in MKII:

```
test {\em test} test
```

This command knows that it switches to italic (or slanted) and when used nested it knows to switch back. It also knows if a bold italic or slanted font is used. Therefore it can add italic correction between an italic and upright shape.



Latin Modern Roman Regular　　Latin Modern Roman Italic

Figure 1: Italic overshoot in Latin Modern.

An italic correction is bound to a glyph and bound to a font. In figure 1 we see how an italic shape extends out of the bounding box. This is not the case in Dejavu: watch figure 2.

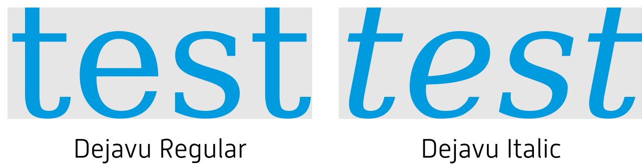


Figure 2: Italic overshoot in Dejavu Serif.

This means that the application of italic correction should never been applied without knowing the font. In figure 3 we see an upright word following an italic. The space is determined by the upright one.

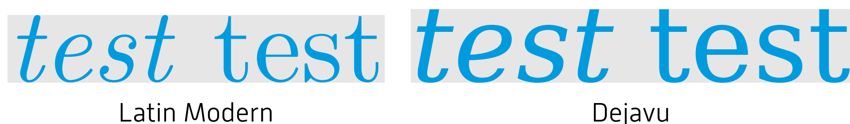


Figure 3: Italic followed by upright.

Because it is to be used with care you need to enable this feature per font, You also need to explicitly enable the application of this correction. In figure 4 we see italic correction in action.

```
\definefontfeature
  [italic]
  [default]
  [itlc=yes]
```

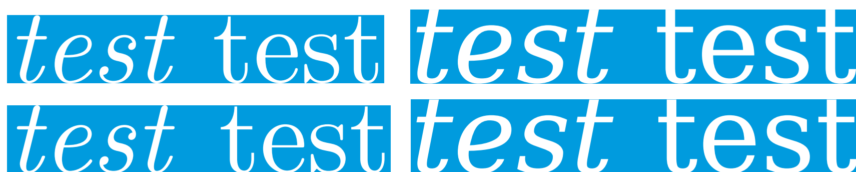


Figure 4: Italic correction.

This only signals the font constructor that additional italic information has to be added to the font metrics. As we already mentioned, the application of correction is driven by the \/ primitive and that one consults the font metrics. Because the correction is not part of the original font metrics it is calculated automatically by adding a small value to the width. This value is calculated as follows:

```
factor * (parameters.uwidth or 40) / 2
```

The `uwidth` parameter is sometimes part of the specification but if not, we take a reasonable default. The factor is under user control:

```
\definefontfeature
  [moreitalic]
  [default]
  [itlc=5]
```

This is demonstrated in figure 5. You will notice that for Latin Modern (any) correction makes sense, but for Dejavu it probably makes things look worse. This is why italic correction is disabled by default. When enabled there are several variants:

global	always apply correction
text	only apply correction to text
always	apply correction between text and boxes
none	forget about correction

We keep track of the state using attributes but that comes at a (small) price in terms of extra memory and runtime. The `global` option simply assumes that we always need to check for correction (of course only for fonts that have this feature enabled). In the given example we used:

```
\setupitaliccorrection
  [text]
```

You can combine keys:

```
\setupitaliccorrection
  [global,always]
```

test test *test* test
test test *test* test

Figure 5: Italic correction (factor 5).

The `itlc` feature controls if a font gets italic correction applied. In principle this is all that the user needs to do, given that the mechanism is enabled. There is an extra feature that controls the implementation:

`itlc`	`no`	don't apply italic correction (default)
	`yes`	apply italic correction
`textitalics`	`no`	precalculate italic corrections (permit engine usage)
	`yes`	precalculate italic corrections (inhibit engine)
	`delay`	delay calculation of corrections

When `textitalics` is set to `yes` or `delay` the mechanism built into the engine is completely disabled. When set to `no` the engine can kick in but normally the alternative method takes precedence so that the engine sees no reason for further action. You can trace italic corrections with:

```
\enabletrackers[typesetters.italics]
```

3. Bounding boxes

There are some features that are rather useless and only make sense when figuring out issues. An example of such a feature is the following:

```
\definefontfeature
  [withbbox]
  [boundingbox=yes]

\definefont
  [FontWithBB]
  [Normal*withbbox]
```

This feature adds a background to each character in a font. In some fonts a glyph has a tight bounding box, while on other fonts some extra space is put on the left and right. Keep in mind that this feature blocks colored text.

4. Math italics

In the traditional TEX fonts the width of a glyph was not the real width because one had to add the italic correction to it. The engine then juggles a bit with these properties. If you run into fonts that are designed this way, you can do this:

```
\definefontfeature[mathextra][italicwidths=yes] % fix latin
modern
```

This might make `$\left|V\right| = \left|W\right|$` look better for such fonts. Of course there can be side effects because these fonts assume a traditional engine.

5. Slanting

These features (as well as the one described in the next section) are seldom used but provided because they were introduced in pdfTEX.

```
\definefontfeature
  [abitslanted]
  [default]
  [slant=.1]

\definefontfeature
  [abitmoreslanted]
  [default]
  [slant=.2]
```

```
\definedfont[Normal*abitslanted]This is a bit slanted.
\definedfont[Normal*abitmoreslanted]And this is a bit more
slanted.
```

The result is:

This is a bit slanted.
And this is a bit more slanted.

6. Extending

The second manipulation is extending the shapes horizontally:

```
\definefontfeature
  [abitbolder]
  [default]
  [extend=1.3]

\definefontfeature
  [abitnarrower]
  [default]
  [extend=0.7]
```

```
\definedfont[Normal*abitbolder]This looks a bit bolder.
\definedfont[Normal*abitnarrower]And this is a bit narrower.
```

The result is:

This looks a bit bolder. And this is a bit narrower.

We can also combine slanting and extending:

```
\definefontfeature
  [abitofboth]
  [default]
  [extend=1.3,
   slant=.1]
```

```
\definedfont[Normal*abitofboth]This is a bit bolder but also
slanted.
```

If you remember those first needle matrix printers you might recognize the next rendering:

This is a bit bolder but also slanted.

7. Fixing

This is a rather special one. First we show a couple of definitions:

```
\definefontfeature
  [dimensions-a]
  [default]
  [dimensions={1,1,1}]

\definefontfeature
  [dimensions-b]
  [default]
  [dimensions={1,2,3}]

\definefontfeature
  [dimensions-c]
  [default]
  [dimensions={1,3,2}]
```

```
\definefontfeature
  [dimensions-d]
  [default]
  [dimensions={3,3,3}]
```

When you don't want a dimension to change you leave an entry empty, so valid entries are for instance: `,3,` and `1,,`.
As usual you apply such a feature as follows:

```
\definefont[MyFont][Serif*dimensions-a sa 2]
```

Alternatively you can use such a feature on its own:

```
\definefontfeature
  [dimensions-333]
  [dimensions={3,3,3}]
\definefont[MyFont][Serif*default,dimensions-333 sa 2]
```

In figure 6 you see these four definitions in action. The leftmost rendering is the default rendering. The three numbers in the definitions represent the width (in em), height and depth (in ex).

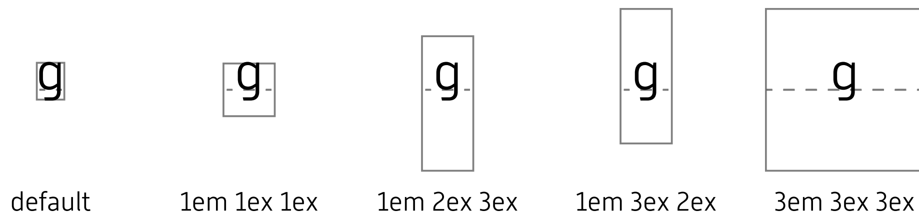


Figure 6: Freezing dimensions of glyphs.

This feature only makes sense for fonts that need a fixed width, like the CJK fonts that are used for asian scripts. Normally those fonts already have fixed dimensions, but this feature can be used to fix problematic fonts or add some more space. However, for such large fonts this also brings a larger memory footprint.
A special case is the following:

```
\definefontfeature
  [dimensions-e]
  [dimensions=strut]
```

This will make the height and depth the same as the *current* strut height and depth:

```
\ruledhbox{\definedfont[Serif*default,dimensions-e at  8pt]clipped}
\ruledhbox{\definedfont[Serif*default,dimensions-e at 12pt]clipped}
\ruledhbox{\definedfont[Serif*default,dimensions-e at 24pt]clipped}
```

The dimensions are (in this case) limited:

Another special case is `dimensions=mono` which will make all characters the font's em-width. This is handy when you define font fallbacks where glyphs come from a non monospaced font.

8. Unicoding

Nowadays we will mostly use fonts that ship with a UNICODE aware encoding. And in ConTEXt, even if we use a TYPE1 font, it gets mapped onto UNICODE. However, there are some exceptions, for instance the Zapf Dingbats in TYPE1 format. These have a rather obscure private encoding and the glyph names run from `a1` up to `a206` and have no relation to what the glyph represents.
In the case of Dingbats we're somewhat lucky that they ended up in UNICODE, so we can relocate the glyphs to match their rightful place. This is done by means of a goodies file.

```
\definefontfeature
  [dingbats]
  [mode=base,
   goodies=dingbats,
   unicoding=yes]

\definefontsynonym
  [ZapfDingbats]
  [file:uzdr.afm]
  [features=dingbats]
```

I tend to qualify the Dingbat font in TEX distributions as rather unstable because of name changes and them either or not being included. Therefore it's best to use the hard coded name because that triggers the most visible error message when the font is not found.
A font like this can for instance be used with the glyph placement macros as is demonstrated below. In the last line we see that a direct UTF input also works out well.

`\getglyphdirect{ZapfDingbats*dingbats}{\number "2701}`	✁	
`\getglyphdirect{ZapfDingbats*dingbats}{\char "2701}`	✁	
`\getnamedglyphdirect{ZapfDingbats*dingbats}{a1}`	✁	
`\getnamedglyphdirect{ZapfDingbats*dingbats}{a11}`	☛	
`\getglyphdirect{ZapfDingbats}{\number "2701}`		unknown
`\getglyphdirect{ZapfDingbats}{\char "2701}`		unknown
`\getnamedglyphdirect{ZapfDingbats}{a1}`	✁	
`\getnamedglyphdirect{ZapfDingbats}{a11}`	☛	
`\definedfont[ZapfDingbats*dingbats]✁`	✁	

Keep in mind that fonts like Dejavu (that we use here as document font) already has these characters which is why it shows up in the verbose part of the table. (The CG Journal uses the Alwyn New font which has a smaller glyph repertoire — red.)

9. Protrusion

Protrusion is a feature that LuaTEX inherits from pdfTEX. It is sometimes referred to as hanging punctuation but in our case any character qualifies. Also, hanging is not frozen but can be tuned in detail. Currently the engine defines protrusion in terms of the emwidth which is unfortunate and likely to change.[1]

It is sometimes believed that protrusion improves for instance narrower columns, but I'm pretty sure that this is not the case. It is true that it is taken into account when breaking a paragraph into lines, and that we then have a little bit more width available, but at the same time it is an extra constraint: if we protrude we have to do it for each line (and the whole main body of text) so it's just a different solution space. The main reason for applying this feature is *not* that the lines look better or that we get better looking narrow lines but that the right and left margins look nicer. Personally I don't like half protrusion of punctuation and hyphens. Best is to have small values for regular characters to improve the visual appearance and use full protrusion for hyphens (and maybe punctuation).

protrusion classes

In ConTEXt we've always defined protrusion as a percentage of the width of a glyph. From MKII we inherit the level of control as well as the ability to define vectors. The shared properties are collected in so called classes and the character specific properties in vectors. The following classes are predefined:

name	**vector**	**factor**	**left**	**right**
alpha	alpha	1.00		
double		2.00	1.00	1.00
preset		1.00	1.00	1.00
punctuation	punctuation	1.00		
pure	pure	1.00		
quality	quality	1.00		

[1] In general the low level implementation can be optimized as there are better mechanisms in LuaTEX.

The names are used in the definitions:

```
\definefontfeature[default][protrusion=quality]
```

Currently adding a class only has a Lua interface.

```
\startluacode
fonts.protrusions.classes.myown = {
    vector = 'myown',
    factor = 1,
}
\stopluacode
```

protrusion vectors

Vectors are larger but not as large as you might expect. Only a subset of characters needs to be defined. This is because in practice only latin scripts are candidates and these scripts have glyphs that look a lot like each other. As we only operate on the horizontal direction characters like ‘aàáâãäå’ look the same from the left and right so we only have to define the protrusion for ‘a’.
As with classes, you can define your own vectors:

```
\startluacode
fonts.protrusions.vectors.myown = table.merged (
    fonts.protrusions.vectors.quality,
    {   [0x002C] = { 0, 2 },   } -- comma
)
\stopluacode
```

protrusion vector pure

U+0002C	0.00	,	1.00	U+000AD	0.00		1.00	U+02014	0.00	—	0.33
U+0002D	0.00	-	1.00	U+0060C	0.00	،	1.00	U+03001	0.00		1.00
U+0002E	0.00	.	1.00	U+0061B	0.00	؛	1.00	U+03002	0.00		1.00
U+0003A	0.00	:	1.00	U+006D4	0.00		1.00				
U+0003B	0.00	;	1.00	U+02013	0.00	–	0.50				

protrusion vector punctuation

U+00021	0.00	!	0.20	U+0002E	0.00	.	0.70	U+0005D	0.00	]	0.05
U+00028	0.05	(	0.00	U+0003A	0.00	:	0.50	U+000A1	0.00	¡	0.20
U+00029	0.00	)	0.05	U+0003B	0.00	;	0.50	U+000AB	0.50	«	0.50
U+0002C	0.00	,	0.70	U+0003F	0.00	?	0.20	U+000AD	0.00		0.70
U+0002D	0.00	-	0.70	U+0005B	0.05	[	0.00	U+000BB	0.50	»	0.50

U+000BF	0.00	¿	0.20	U+02014	0.00	—	0.20	U+0201D	0.00	”	0.50
U+0060C	0.00	،	0.70	U+02018	0.70	‘	0.70	U+0201E	0.50	„	0.00
U+0061B	0.00	؛	0.50	U+02019	0.00	’	0.70	U+0201F	0.50	‟	0.00
U+0061F	0.00	؟	0.20	U+0201A	0.70	‚	0.00	U+02039	0.70	‹	0.70
U+006D4	0.00		0.70	U+0201B	0.70	‛	0.00	U+0203A	0.70	›	0.70
U+02013	0.00	–	0.30	U+0201C	0.50	“	0.50				

protrusion vector alpha

U+00041	0.05	A	0.05	U+00056	0.05	V	0.05	U+00074	0.00	t	0.05
U+00046	0.00	F	0.05	U+00057	0.05	W	0.05	U+00076	0.05	v	0.05
U+0004A	0.05	J	0.00	U+00058	0.05	X	0.05	U+00077	0.05	w	0.05
U+0004B	0.00	K	0.05	U+00059	0.05	Y	0.05	U+00078	0.05	x	0.05
U+0004C	0.00	L	0.05	U+0006B	0.00	k	0.05	U+00079	0.05	y	0.05
U+00054	0.05	T	0.05	U+00072	0.00	r	0.05				

protrusion vector quality

U+00021	0.00	!	0.20	U+00058	0.05	X	0.05	U+0061B	0.00	؛	0.50
U+00028	0.05	(	0.00	U+00059	0.05	Y	0.05	U+0061F	0.00	؟	0.20
U+00029	0.00	)	0.05	U+0005B	0.05	[	0.00	U+006D4	0.00		0.70
U+0002C	0.00	,	0.70	U+0005D	0.00	]	0.05	U+02013	0.00	–	0.30
U+0002D	0.00	-	0.70	U+0006B	0.00	k	0.05	U+02014	0.00	—	0.20
U+0002E	0.00	.	0.70	U+00072	0.00	r	0.05	U+02018	0.70	‘	0.70
U+0003A	0.00	:	0.50	U+00074	0.00	t	0.05	U+02019	0.00	’	0.70
U+0003B	0.00	;	0.50	U+00076	0.05	v	0.05	U+0201A	0.70	‚	0.00
U+0003F	0.00	?	0.20	U+00077	0.05	w	0.05	U+0201B	0.70	‛	0.00
U+00041	0.05	A	0.05	U+00078	0.05	x	0.05	U+0201C	0.50	“	0.50
U+00046	0.00	F	0.05	U+00079	0.05	y	0.05	U+0201D	0.00	”	0.50
U+0004A	0.05	J	0.00	U+000A1	0.00	¡	0.20	U+0201E	0.50	„	0.00
U+0004B	0.00	K	0.05	U+000AB	0.50	«	0.50	U+0201F	0.50	‟	0.00
U+0004C	0.00	L	0.05	U+000AD	0.00		0.70	U+02039	0.70	‹	0.70
U+00054	0.05	T	0.05	U+000BB	0.50	»	0.50	U+0203A	0.70	›	0.70
U+00056	0.05	V	0.05	U+000BF	0.00	¿	0.20				
U+00057	0.05	W	0.05	U+0060C	0.00	،	0.70				

examples of protrusion

Next we show the quality protrusion. For this we use `tufte.tex` as this one for sure will result in punctuation and other candidates for protrusion.

```
\definefontfeature
  [whatever][default]
  [protrusion=quality]

\definefont[MyTestA][Serif*default  at 10pt]
\definefont[MyTestB][Serif*whatever at 10pt]
```

We thrive in information--thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

Figure 7: The difference between no protrusion and quality protrusion.

We use the following example. The results are shown in figure 7. The colored text is the protruding one.

```
\startoverlay
    {\ruledvbox \bgroup
        \hsize\textwidth
        \MyTestA
        \setupalign[normal]
        \input{tufte}
     \egroup}
    {\ruledvbox \bgroup
        \hsize\textwidth
        \MyTestB
        \setupalign[hanging,normal]
        \maincolor
        \input{tufte}
     \egroup}
\stopoverlay
```

The previously defined own class and vector is somewhat more extreme:

```
\definefontfeature
  [whatever]
  [default]
  [protrusion=myown]

\definefont[MyTestA][Serif*default  at 10pt]
\definefont[MyTestB][Serif*whatever at 10pt]
```

In figure 8 we see that the somewhat extreme definition of the comma also pulls the preceding character into the margin.

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

Figure 8: The influence of extreme protrusion on preceding characters.

10. Expansion

Expansion is also an inheritance of pdfTEX.[2] This mechanism selectively expands characters, normally up to 5%. One reason for applying it is that we have less visually incompatible spacing, especially when we have underfull or cramped lines. For each (broken) line the badness is reconsidered with either shrink or stretch applied to all characters in that line. So, in the worst case a shrunken line is followed by a stretched one and that can be visible when the scaling factors are chosen wrongly.
As with protrusion, the solution space is larger but so are the constraints. But contrary to protrusion here the look and feel of the whole line can be made better but at the cost of much more runtime and larger (PDF) files.

protrusion classes

The amount of expansion depends in the shape of the character. Vertical strokes are more sensitive for expansion than horizontal ones. So an 'o' can get a different scaling than an 'm'. As with protrusion we have collected the properties in classes:

name	vector	step	factor	stretchshrink
preset		0.50	1.00	22
quality	default	0.50	1.00	22

The smaller the step, the more instances of a font we get, the better it looks, and the larger the files become. It is best not to use too many stretch and shrink steps. A stretch of 2 and shrink of 2 and step of .25 results in up to 8 instances plus the regular sized one.

expansion vectors

We only have one vector: `quality`:

U+00032	2	0.70	U+00041	A	0.50	U+00046	F	0.70
U+00033	3	0.70	U+00042	B	0.70	U+00047	G	0.50
U+00036	6	0.70	U+00043	C	0.70	U+00048	H	0.70
U+00038	8	0.70	U+00044	D	0.50	U+0004B	K	0.70
U+00039	9	0.70	U+00045	E	0.70	U+0004D	M	0.70

[2] As with protrusion the implementation in the engine is somewhat suboptimal and inefficient and will be upgraded to a more LuaTEX-ish way.

U+0004E	N	0.70	U+00061	a	0.70	U+0006E	n	0.70
U+0004F	O	0.50	U+00062	b	0.70	U+0006F	o	0.70
U+00050	P	0.70	U+00063	c	0.70	U+00070	p	0.70
U+00051	Q	0.50	U+00064	d	0.70	U+00071	q	0.70
U+00052	R	0.70	U+00065	e	0.70	U+00073	s	0.70
U+00053	S	0.70	U+00067	g	0.70	U+00075	u	0.70
U+00055	U	0.70	U+00068	h	0.70	U+00077	w	0.70
U+00057	W	0.70	U+0006B	k	0.70	U+0007A	z	0.70
U+0005A	Z	0.70	U+0006D	m	0.70			

an example of expansion

We use `zapf.tex` as example text, if only because Hermann Zapf introduced this optimization. Keep in mind that you can combine expansion and protrusion.

```
\definefontfeature
  [whatever]
  [default]
  [expansion=quality]

\definefont[MyTestA][Serif*default  at 10pt]
\definefont[MyTestB][Serif*whatever at 10pt]
```

We use the following example. The results are shown in figure 9. The colored text is the protruding one.

```
\startoverlay
    {\ruledvbox \bgroup
        \hsize\textwidth
        \MyTestA
        \setupalign[normal]
        \input{tufte}
     \egroup}
    {\ruledvbox \bgroup
        \hsize\textwidth
        \MyTestB
        \setupalign[hz,normal]
        \maincolor
        \input{tufte}
     \egroup}
\stopoverlay
```

You can see what happens in figure 10.

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

Figure 9: The difference between no expansion and quality expansion.

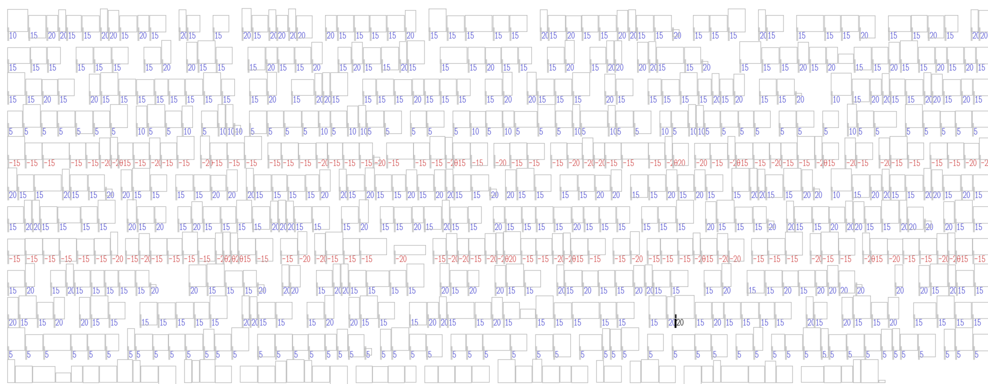


Figure 10: The injected expansion kerns.

```
\setupalign[hz]
\enabletrackers[*expansion*]
\definefontfeature[boundingbox][boundingbox={frame,empty}]
\definedfont[Serif*default,quality,boundingbox @ 12.1pt]
\samplefile{sapolsky}\par
\disabletrackers[*expansion*]
```

Expansion and kerning

When we expand glyphs we also need to look at the font kerns between them. In the original implementation taken from pdfTEX expansion was implemented using pseudo fonts (with expanded glyph widths) and expansion of inter-character kerns was based on font information. In LuaTEX we have expansion factors in glyph nodes instead which is more efficient and gives a cleaner separation between front- and backend as the backend has no need to consult the font.

For the font kerns we set the kern compensation directly and for that we use the average expansion factors of the neighbouring fonts so technically we support kerns between different fonts). This also has the advantage that kerns injected in node mode are treated well, given that they are tagged as font kern.

So what is the effect (and need) of scaling font kerns? Let's look at an example. Kerns can be positive but also negative:

The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

no hz & hz

The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

no hz & full hz

The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

hz & full hz

Figure 11: The two expansion methods compared.

VA II
VA II

negative positive

If we use a ridiculous amount of stretch we get the following. In the top line we scale the kern, in the bottom line we don't.

VA II
VA II

negative positive

The reason that we mention this is that when we apply OPENTYPE features, positioning not necessarily results in font kerns. For instance ligatures can be the result of careful applied kerns and in some scripts kerns are used to connect glyphs. This means that we best cannot expand kerns by default. How bad is that? By looking at the examples above one would say "really bad".

But say that we have about 1pt of font kerns, then a 5% expansion (which is already a lot) amounts to 0.05pt so to | we add | which is so little that it probably goes unnoticed. Even if we use extreme kerns, as between VA, in practice the small amount of stretch or shrink added to a font kern goes unnoticed.

In figure 11 we have overlayed the different strategies. The sample and width is chosen such that we see something. On a display you can scale up these examples and inspect if there is really something to see, but on paper zooming in helps, as in figure 12. Even then the effect of expanded kerns is invisible. The used definitions are:

```
\setupfontexpansion [extremehz]
    [stretch=5,shrink=5,step=.5,vector=default,factor=1]
\setupfontexpansion [regularhz]
    [stretch=2,shrink=2,step=.5,vector=default,factor=1]
\setupfontexpansion [minimalhz]
    [stretch=2,shrink=2,step=.5,vector=default,factor=.5]

\definefontfeature [extremehz] [default]
    [mode=node,expansion=extremehz]
\definefontfeature [regularhz] [default]
    [mode=node,expansion=regularhz]
\definefontfeature [minimalhz] [default]
    [mode=node,expansion=minimalhz]

\definefont
    [ExtremeHzFont]
    [file:texgyrepagella-regular.otf*extremehz at 10pt]
\definefont
    [RegularHzFont]
    [file:texgyrepagella-regular.otf*regularhz at 10pt]
\definefont
    [MinimalHzFont]
    [file:texgyrepagella-regular.otf*minimalhz at 10pt]
```

eral, in fact. It would	eral, in fact. It would	eral, in fact. It would
extreme: no hz & hz	extreme: no hz & full hz	extreme: hz & full hz
eral, in fact. It would	eral, in fact. It would	eral, in fact. It would
regular: no hz & hz	regular: no hz & full hz	regular: hz & full hz
eral, in fact. It would	eral, in fact. It would	eral, in fact. It would
minimal: no hz & hz	minimal: no hz & full hz	minimal: hz & full hz

Figure 12: The two expansion methods compared (zoomed in).

In ConTEXt the `hz` alignment option only enables expansion of glyphs, while `fullhz` also applies it to kerns. It will be clear that you can just stick to using the `hz` directive (if you want expansion at all) because this directive is normally disabled and because most fonts are processed in node mode.

11. Composing

This feature is seldom needed but can come in handy for old fonts or when some special language is to be supported. When writing this section I tested this feature with Dejavu and only two additional characters were added:

```
fonts > combining > Ѷ (U+00476) = Ѵ (U+00474) + ̏ (U+0030F)
fonts > combining > ѷ (U+00477) = ѵ (U+00475) + ̏ (U+0030F)
```

This trace showed up after giving:

```
\enabletrackers
  [fonts.composing.define]

\definefontfeature
  [default-plus-compose]
  [compose=yes]

\definefont
  [MyFont]
  [Serif*default-plus-compose]
```

Fonts like Latin Modern have lots of glyphs but still lack some. Although the composer can add some of the missing, some of those new virtual glyphs probably will never look really good. For instance, putting additional accents on top of already accented uppercase characters will fail when that character has a rather tight (or even clipped) boundingbox in order not to spoil the lineheight. You can get some more insight in the process by turning on tracing:

```
\enabletrackers[fonts.composing.visualize]
```

One reason why composing can be suboptimal is that it uses the boundingbox of the characters that are combined. If you really depend on a specific font and need some of the missing characters it makes sense to spend some time on optimizing the rendering. This can be done via the goodies mechanism. As an example we've added `lm-compose-test.lfg` to the distribution. First we show how it looks at the TEX end:

```
\enabletrackers[fonts.composing.visualize]

\definefontfeature
  [default-plus-compose]
  [compose=yes]
```

```
\loadfontgoodies
  [lm-compose-test] % playground

\definefont
  [MyComposedSerif]
  [file:lmroman10regular*default-plus-compose at 48pt]
```

B Ḃ Ḅ

The positions of the dot accents on top and below the capital B is defined in a goodie file:

```
return {
  name = "lm-compose-test",
  version = "1.00",
  comment = "Goodies that demonstrate composition.",
  author = "Hans and Mojca",
  copyright = "ConTeXt development team",
  compositions = {
    ["lmroman12-regular"] = compose,
  }
}
```

As this is an experimental feature there are several ways to deal with this. For instance:

```
local defaultfraction = 10.0

local compose = {
  dy       = defaultfraction,
  [0x1E02] = { -- B dot above
      dy = 150
  },
  [0x1E04] = { -- B dot below
      dy = 150
  },
}
```

Here the fraction is relative to the difference between the height of the accentee and the accent. A better solution is the following:

```
local compose = {
  [0x1E02] = { -- B dot above
    anchored = "top",
  },
  [0x1E04] = { -- B dot below
    anchored = "bottom",
  },
  [0x0042] = { -- B
    anchors = {
      top = {
        x = 300, y = 700,
      },
      bottom = {
        x = 300, y = -30,
      },
    },
  },
  [0x0307] = {
    anchors = {
      top = {
        x = -250, y =  550,
      },
    },
  },
  [0x0323] = {
    anchors = {
      bottom = {
        x = -250, y =  -80,
      },
    },
  },
}
```

This approach is more or less the same as OpenType anchoring. It takes a bit more effort to define these tables but the result is better.

12. Kerning

Inter-character kerning is not supported at the font level and with good reason. The fact that something is conceptually possible doesn't mean that we should use or support it. Normally proper kerning (or the lack of it) is part of a font design and for some scripts different kerning is not even an option.

On the average TEX does a proper job on justification but not all programs are that capable. As a consequence designers (at least we ran into it) tend to stick to flush left rendering because they don't trust their system to do a proper job otherwise. On the other hand they seem to have no problem with messing up the inter-character spacing

and even combine that with excessive inter-word spacing *if* they want to achieve justification (without hyphenation). And it can become even worse when extreme glyph expansion (like hz) is applied.
Anyhow, it will be clear that consider messing with properties like kerning that are part of the font design is to be done carefully.
For running text additional kerning makes no sense. It not only looks bad, it also spoils the grayness of a text. When it is applied we need to deal with special cases. For instance ligatures make no sense so they should be disabled. Additional kerning should relate to already present kerning and interword spacing should be adapted accordingly. Embedded non-characters also need to be treated well.
This paragraph was typeset as follows:

```
\definecharacterkerning [extremekerning] [factor=.125]

\setcharacterkerning[extremekerning] ... text ...
```

Where additional kerning can make sense, is in titles. The previous command can do that job. In addition we have a mechanism that fills a given space. This mechanism uses the following definition:

```
\setupcharacterkerning
  [stretched]
  [factor=max,
   width=\availablehsize]
```

```
\stretched{\bfd to the limit}
```

t o t h e l i m i t

The following does not work:

```
\ruledhbox to 5cm{\stretched{\bfd to the limit}}
```

t o t h e l i m i t

But this works ok:

```
\setupcharacterkerning
  [stretched]
  [width=]

\stretched{\bfd to the limit}
```

t o t h e l i m i t

You can also say this:

```
\stretched[width=]{\bfd to the limit}
```

t o t h e l i m i t

or:

```
\ruledhbox{\stretched[width=10cm]{\bfd to the limit}}
```

t o t h e l i m i t

You can get some insight in what kerning does to your font by the following command:

```
\usemodule[typesetting-kerning]

\starttext
  \showcharacterkerningsteps
    [style=Bold,
     sample=how to violate a proper font design,
     text=rubish,
     first=0,
     last=45,
     step=5]
\stoptext
```

factor	sample	%	text	%
0.000	how to violate a proper font design how to violate a proper font design	0.00	rubish rubish	0.00

0.005	how to violate a proper font design	0.92	rubish	0.92
	how to violate a proper font design		rubish	
0.010	how to violate a proper font design	1.82	rubish	1.82
	how to violate a proper font design		rubish	
0.015	how to violate a proper font design	2.70	rubish	2.71
	how to violate a proper font design		rubish	
0.020	how to violate a proper font design	3.57	rubish	3.58
	how to violate a proper font design		rubish	
0.025	how to violate a proper font design	4.42	rubish	4.43
	how to violate a proper font design		rubish	
0.030	how to violate a proper font design	5.26	rubish	5.27
	how to violate a proper font design		rubish	
0.035	how to violate a proper font design	6.08	rubish	6.10
	how to violate a proper font design		rubish	
0.040	how to violate a proper font design	6.89	rubish	6.91
	how to violate a proper font design		rubish	
0.045	how to violate a proper font design	7.69	rubish	7.71
	how to violate a proper font design		rubish	

13. Extra font kerns

Fonts are processed independent of each other. Sometimes that is unfortunate for kerning, although in practice it won't happen that often. We can enable an additional kerning mechanism to deal with these cases. The `\setextrafontkerns` command takes one argument between square brackets. The effect can be seen below:

key	result	logic
no kerns	VA VA VA VA **VA** V**A** **V**A V*A*	no kerns at all
kerns	VA VA VA VA **VA** V**A** **V**A V*A*	kerns within a font (feature) run
none	VA VA VA VA **VA** V**A** **V**A V*A*	only extra kerns within fonts
min	VA VA VA VA **VA** V**A** **V**A V*A*	minimal kerns within and across fonts
max	VA VA VA VA **VA** V**A** **V**A V*A*	maximum kerns within and across fonts
mixed	VA VA VA VA **VA** V**A** **V**A V*A*	averaged kerns within and across fonts

The content is defined as:

```
VA {\smallcaps va} V{\smallcaps a}
VA {\bf VA} V{\bf A} {\bf V}A
V{\it A}
```

This mechanism obeys grouping so you have complete control over where and when it gets applied. The `\showfontkerns` command can be used to trace the injection of (font) kerns.

14. Ligatures

For some Latin fonts ligature building is quite advanced, take Unifraktur. I have no problem admitting that I find fraktur hard to read, but this one actually is sort of an exception. It's also a good candidate for a screen presentation where you mainly made notes for yourself: no one has to read it, but it looks great, especially if you consider it to be drawn by a pen.
Anyway, we will use the following code as example (based on some remarks on the fonts website).

```
sitzen / ſitzen / effe fietsen / ch ck ſt tz ſi fi
```

Some ligatures are implemented in the usual way, using the `liga` and `dlig` features, others kick in thanks to `ccmp`. This fact alone is an illustration that the low level OpenType ligature feature is not related to ligatures at all but a more generic mechanism: you can basically combine multiple shapes into one in all features exposed to the user.
We define a bunch of specific feature sets:

```
\definefontfeature
  [unifraktur-a]
  [default]
\definefontfeature
  [unifraktur-b]
  [default]
  [goodies=unifraktur,keepligatures=yes]
\definefontfeature
  [unifraktur-c]
  [default]
  [ccmp=yes]
\definefontfeature
  [unifraktur-d]
  [default]
  [ccmp=yes,goodies=unifraktur,keepligatures=yes]
\definefontfeature
  [unifraktur-e]
  [default]
  [liga=no,rlig=no,clig=no,dlig=no,ccmp=yes,keepligatures=auto]
```

and also some fonts:

```
\definefont[TestA][UnifrakturCook*unifraktur-a sa 0.9]
\definefont[TestB][UnifrakturCook*unifraktur-b sa 0.9]
\definefont[TestC][UnifrakturCook*unifraktur-c sa 0.9]
\definefont[TestD][UnifrakturCook*unifraktur-d sa 0.9]
\definefont[TestE][UnifrakturCook*unifraktur-e sa 0.9]
```

We show these five alternatives here:

liga	sitzen / ſitzen / effe fietsen / ch ck ſt tz ſi fi
liga + keepligatures	sitzen / ſitzen / effe fietsen / ch ck ſt tz ſi fi
liga + ccmp	sitzen / ſitzen / effe fietsen / ch ck ſt tz ſi fi
liga + ccmp + keepligatures	sitzen / ſitzen / effe fietsen / ch ck ſt tz ſi fi
ccmp + keepligatures	sitzen / ſitzen / effe fietsen / ch ck ſt tz ſi fi

The real fun starts when we want to add extra spacing between characters. Some ligatures need to get broken and some kept.

```
\setupcharacterkerning[kerncharacters][factor=0.5]
\setupcharacterkerning[letterspacing] [factor=0.5]
```

Next we will see how ligatures behave depending on how the mechanisms are set up. The colors indicate what trickery is used:

red	kept by dynamic feature
green	kept by static feature
blue	keep by goodie

First we use `\kerncharacters`:

liga

s i t z e n / ſ i t z e n / e f f e f i e t s e n / c h c k ſ t t z ſ i f i

liga + keepligatures

s i t z e n / ſ i t z e n / e f f e f i e t s e n / c h c k ſ t t z ſ i f i

liga + ccmp

s i t z e n / ſ i t z e n / e f f e f i e t s e n / c h c k ſ t t z ſ i f i

liga + ccmp + keepligatures

s i t z e n / ſ i t z e n / e f f e f i e t s e n / c h c k ſ t t z ſ i f i

ccmp + keepligatures

s i t z e n / ſ i t z e n / e f f e f i e t s e n / c h c k ſ t t z ſ i f i

In the next example we use `\letterspacing`:

liga

s i t z e n / ſ i t z e n / e f f e f i e t s e n / c h c k ſ t t z ſ i f i

liga + keepligatures

```
          s i t z e n / ſ i t z e n / e f f e f i e t s e n / ch ck ſ t t z ſ i f i
liga + ccmp
          s i t z e n / ſ i t z e n / e f f e f i e t s e n / ch ck ſ t t z ſ i f i
liga + ccmp + keepligatures
          s i t z e n / ſ i t z e n / e f f e f i e t s e n / ch ck ſ t t z ſ i f i
ccmp + keepligatures
          s i t z e n / ſ i t z e n / e f f e f i e t s e n / ch ck ſ t t z ſ i f i
```

The difference is that the letterspacing variant dynamically adds the predefined featureset `letterspacing` which is defined in a similar way as `unifraktur-e`. In the case of this font, this variant is the better one to use. In fact, this variant probably works okay with most fonts. However, by not hard coding this behaviour we keep control, as one never knows what the demands are. When no features are used, information from the (given) goodie file `unifraktur.lfg` is consulted:

```
letterspacing = {
  -- watch it: zwnj's are used (in the tounicodes too)
  keptligatures = {
    ["c_afii301_k.ccmp"] = true, -- ck
    ["c_afii301_h.ccmp"] = true, -- ch
    ["t_afii301_z.ccmp"] = true, -- tz
    ["uniFB05"]          = true, -- ﬅ
  },
}
```

These kick in when we don't disable ligatures by setting features (case e).
There are two pseudo features that can help us out when a font doesn't provide the wanted ligatures but has the right glyphs for building them. The UNICODE database has some information about how characters can be (de)composed and we can use that information to create virtual glyphs:

```
\definefontfeature
  [default] [default]
  [char-ligatures=yes,mode=node]
```

and:

```
\definefontfeature
  [default] [default]
  [compat-ligatures=yes,mode=node]
```

This feature was added after some discussion on the ConTEXt mailing list about the following use case.

```
\definefontfeature
  [default-l] [default]
  [char-ligatures=yes,
   compat-ligatures=yes,
   mode=node]

\definefont[LigCd][cambria*default]
\definefont[LigPd][texgyrepagellaregular*default]
\definefont[LigCl][cambria*default-l]
\definefont[LigPl][texgyrepagellaregular*default-l]
```

These definitions result in:

	`\LigCd`	`\LigPd`	`\LigCl`	`\LigPl`
PEL·LÍCULES	PEL·LÍCULES	PEL·LÍCULES	PEĿLÍCULES	PEĿLÍCULES
pel·lícules	pel·lícules	pel·lícules	peŀlícules	peŀlícules
PEĿLÍCULES	PEĿLÍCULES	PEĿLÍCULES	PEĿLÍCULES	PEĿLÍCULES
peŀlícules	peŀlícules	peŀlícules	peŀlícules	peŀlícules

Of course one can wonder is this is the right approach and if it's not better to use a font that provides the needed characters in the first place.

15. New features

15.1 Substitution

It is possible to add new features via Lua. Here is an example of a single substitution:

```
\startluacode
    fonts.handlers.otf.addfeature {
        name = "stest",
        type = "substitution",
        data = {
            a = "X",
            b = "P",
        }
    }
\stopluacode
```

We show an overview at the end of this section, but here is a simple example already. You need to define the feature before defining a font because otherwise the font will not know about it.

```
\definefontfeature[stest][stest=yes]
\definedfont[file:dejavu-serifbold.ttf*default]
abracadabra: \addff{stest}abracadabra
```

abracadabra: XPrXcXdXPrX

Instead of (more readable) glyph names you can also give UNICODE numbers:

```
\startluacode
    fonts.handlers.otf.addfeature {
        name = "stest",
        type = "substitution",
        data = {
            [0x61] = 0x58
            [0x62] = 0x50
        }
    }
\stopluacode
```

The definition is quite simple: we just map glyph names (or unicodes) onto other ones. An alternate is also possible:

```
\startluacode
    fonts.handlers.otf.addfeature {
        name = "atest",
        type = "alternate",
        data = {
            a = { "X", "Y" },
            b = { "P", "Q" },
        }
    }
\stopluacode
```

Less useful is a multiple substitution. Normally this one is part of a chain of replacements.

```
\startluacode
    fonts.handlers.otf.addfeature {
        name = "mtest",
```

```
        type = "multiple",
        data = {
            a = { "X", "Y" },
            b = { "P", "Q" },
        }
    }
\stopluacode
```

A ligature (or multiple to one) is also possible but normally only makes sense when there is indeed a ligature. We use a similar definition for mapping the TEX input sequence --- onto an —.

```
\startluacode
    fonts.handlers.otf.addfeature {
        name = "ltest",
        type = "ligature",
        data = {
            ['1'] = { "a", "b" },
            ['2'] = { "d", "a" },
        }
    }
\stopluacode
```

15.2 Positioning

You can define a kern feature too but when doing so you need to use measures in font units.

```
\startluacode
    fonts.handlers.otf.addfeature {
        name = "ktest",
        type = "kern",
        data = {
            a = { b = -500 },
        }
    }
\stopluacode
```

Pairwise positioning is more complex and involves two (optional) arrays that specify `{dx dy wd ht}` for each of the two glyphs. In the next example we only displace the second glyph.

```
\startluacode
    fonts.handlers.otf.addfeature {
        name = "ptest",
        type = "pair",
        data = {
            ["a"] = { ["b"] = { false,  { -1000, 1200, 0, 0 } }
},
        }
    }
\stopluacode
```

Of course you need to know a bit about the metrics of the glyphs involved so in practice this boils down to trial and error.
A single character (glyph) can also be tweaked, although normally this is done better in a manipulator when loading the font. Anyway:

```
\startluacode
    fonts.handlers.otf.addfeature {
        name = "stest",
        type = "single",
        data = {
            a = { -30, 0, -50, 0 },
        }
    }
\stopluacode
```

This will reduce the left and right edges and make the glyph a pretty tight one. The values are for Latin Modern.

15.3 Examples

We didn't show usage yet. This is because we need to define a feature before we define a font. New features will be added to a font when it gets defined.

```
\definefontfeature[stest][stest=yes]
\definefontfeature[atest][atest=2]
\definefontfeature[mtest][mtest=yes]
\definefontfeature[ltest][ltest=yes]
\definefontfeature[ktest][ktest=yes]
\definefontfeature[ptest][ptest=yes]
\definefontfeature[ctest][ctest=yes]

\definedfont[file:dejavu-serif.ttf*default]
```

```
\starttabulate[|l|l|l|]
\NC operation   \NC feature     \NC             abracadabra\NC\NR
\HL
\NC substitution\NC\type {stest}\NC\addff{stest}abracadabra\NC\NR
\NC alternate   \NC\type {atest}\NC\addff{atest}abracadabra\NC\NR
\NC multiple    \NC\type {mtest}\NC\addff{mtest}abracadabra\NC\NR
\NC ligature    \NC\type {ltest}\NC\addff{ltest}abracadabra\NC\NR
\NC kern        \NC\type {ktest}\NC\addff{ktest}abracadabra\NC\NR
\NC pair        \NC\type {ptest}\NC\addff{ptest}abracadabra\NC\NR
\NC chain sub   \NC\type {ctest}\NC\addff{ctest}abracadabra\NC\NR
\stoptabulate
```

operation	feature	abracadabra
substitution	`stest`	abracadabra
alternate	`atest`	YQrYcYdYQrY
multiple	`mtest`	XYPQrXYcXYdXYPQrXY
ligature	`ltest`	1raca2bra
kern	`ktest`	abracadabra
pair	`ptest`	a b racada b ra
chain sub	`ctest`	abracadabra

15.4 Contexts

A more complex substitution is the following:

```
\startluacode
    fonts.handlers.otf.addfeature {
        name    = "ytest",
        type    = "chainsubstitution",
        lookups = {
            {
                type = "substitution",
                data = {
                    ["b"] = "B",
                    ["c"] = "C",
                },
            },
        },
        data = {
            rules = {
                {
                    before  = { { "a" } },
                    current = { { "b", "c" } },
```

```
                    lookups = { 1 },
                },
            },
        },
    }
\stopluacode
```

Here the dataset is a sequence of rules. There can be a `before`, `current` and `after` match. The replacements are specified with the `lookups` entry and the numbers are indices in the provided `lookups` table.
Here is another example. This one demonstrates that one can check against spaces (some fonts kerns against them) and against boundaries as well. The latter is something ConTEXt specific. First we define a feature that create ligatures but only when we touch a space:

```
\startluacode
    fonts.handlers.otf.addfeature {
        name    = "test-a",
        type    = "chainsubstitution",
        lookups = {
            {
                type = "ligature",
                data = {
                    ['1'] = { "a", "b" },
                    ['2'] = { "c", "d" },
                },
            },
        },
        data = {
            rules = {
                {
                    before  = { { " " } },
                    current = { { "a" }, { "b" } },
                    lookups = { 1 },
                },
                {
                    current = { { "c" }, { "d" } },
                    after   = { { " " } },
                    lookups = { 1 },
                },
            },
        },
    }

\stopluacode
```

The next example also checks against whatever boundary we have.

```
\startluacode
    fonts.handlers.otf.addfeature {
        name    = "test-b",
        type    = "chainsubstitution",
        lookups = {
            {
                type = "ligature",
                data = {
                    ['1'] = { "a", "b" },
                    ['2'] = { "c", "d" },
                },
            },
        },
        data = {
            rules = {
                {
                    before  = { { " ", 0xFFFC } },
                    current = { { "a" }, { "b" } },
                    lookups = { 1 },
                },
                {
                    current = { { "c" }, { "d" } },
                    after   = { { 0xFFFC, " " } },
                    lookups = { 1 },
                },
            },
        },
    }
\stopluacode
```

We can actually simplify this one to:

```
\startluacode
    fonts.handlers.otf.addfeature {
        name    = "test-c",
        type    = "chainsubstitution",
        lookups = {
            {
                type = "ligature",
                data = {
                    ['1'] = { "a", "b" },
                    ['2'] = { "c", "d" },
                },
            },
        },
        data = {
            rules = {
                {
                    before  = { { 0xFFFC } },
                    current = { { "a" }, { "b" } },
                    lookups = { 1 },
                },
                {
                    current = { { "c" }, { "d" } },
                    after   = { { 0xFFFC } },
                    lookups = { 1 },
                },
            },
        },
    }
\stopluacode
```

As a bonus we show how to do more complex things:

```
\startluacode
    fonts.handlers.otf.addfeature {
        name    = "test-d",
        type    = "chainsubstitution",
        lookups = {
            {
                type = "substitution",
                data = { ["a"] = "A",
                         ["b"] = "B",
                         ["c"] = "C",
                         ["d"] = "D", },
            },
            {
                type = "ligature",
                data = { ['1'] = { "a", "b" },
                         ['2'] = { "c", "d" }, },
            },
        },
        data = {
            rules = {
                {
                    before  = { { 0xFFFC } },
                    current = { { "a" }, { "b" } },
                    lookups = { 2 },
                },
                {
                    current = { { "c" }, { "d" } },
                    after   = { { 0xFFFC } },
                    lookups = { 2 },
                },
                {
                    current = { { "a" } },
                    after   = { { "b" } },
                    lookups = { 1 },
                },
                {
                    current = { { "c" } },
                    after   = { { "d" } },
                    lookups = { 1 },
                },
            },
        },
    }
\stopluacode
```

With the test text:

```
ababababcdcd ababababcdcd ababababcdcd
```

These four result in:

ababababcdcd ababababcdcd ababababcdcd
ababababcdcd ababababcdcd ababababcdcd
ababababcdcd ababababcdcd ababababcdcd
ababababcdcd ababababcdcd ababababcdcd

15.5 Language dependencies

When OpenType was not around we only had to deal with ligatures, smallcaps and oldstyle and of course kerns. Their number was so small that the term 'features' was not even used. In practice one just loaded a font that had oldstyle or smallcaps or none of that and was done. There were different fonts and sold separately.
In OpenType we have more variation and although these fonts can be much more advanced the lack of standardization (for instance what gets initialized, or what shapes are in the default slots) can lead to messy setups. Some fonts bind features to scripts, some don't, which means that:

```
\definefontfeature[smallcaps][smcp=yes,script=dflt]
\definefontfeature[smallcaps][smcp=yes,script=latn]
\definefontfeature[smallcaps][smcp=yes,script=cyrl]
```

are in fact different and you don't know in advance if you need to specify `dflt` or `latn`. In practice for a feature like smallcaps there is no difference between languages, but for ligatures there can be.
When we extend an existing feature we can think of:

```
\definefontfeature[smallcaps][default][smcp=yes,script=auto]
\definefontfeature[smallcaps][default][smcp=yes,script=*]
```

but that can have side effects too (for instance disabling language specific features). The easiest way to explore this language dependency is to make a feature of our own.

```
\startluacode
fonts.handlers.otf.addfeature {
    name     = "simplify",
    type     = "multiple",
    prepend  = true,
    features = {
        ["*"] = {
            ["deu"] = true
        }
    },
    data     = {
        [utf.byte("ä")] = { "a", "e" },
        [utf.byte("Ä")] = { "A", "E" },
        [utf.byte("ü")] = { "u", "e" },
        [utf.byte("Ü")] = { "U", "E" },
        [utf.byte("ö")] = { "o", "e" },
        [utf.byte("Ö")] = { "O", "E" },
        [utf.byte("ß")] = { "s", "z" },
        [utf.byte("")] = { "S", "Z" }, -- "SHARP S" not in tt
font, red.
    },
}
\stopluacode
```

Here we implement a language specific feature that we use at the TEX end:

```
\definefontfeature
  [simplify-de]
  [simplify=yes,
   language=deu]
```

that we can use as:

```
\definedfont[dejavu-sans*default,simplify-de]%
äüöß
{\de äüöß}
{\nl äüöß}
```

and get: aeueoesz aeueoesz aeueoesz, but as you see, both German and Dutch get the same treatment, which might not be what you want, because in Dutch the diaeresis has a different meaning.

```
\definedfont[dejavu-sans*default]%
                        äüöß
{\de\addff{simplify-de}äüöß}
{\nl                    äüöß}
```

The above restricts the usage so now we get: äüöß aeueoesz äüöß, which is more language bound. You don't need much imagination for extending this:

```
\definefontfeature
  [simplify]
  [simplify=yes,
   language=deu]
```

So what do we expect with the next?

```
\definedfont[dejavu-sans*default]%
                     äüöß
{\de\addff{simplify}äüöß}
{\nl\addff{simplify}äüöß}
```

We get: äüöß aeueoesz aeueoesz, and we see that the language setting is not taken into account! This is because the font already has been set up with a script and language combination. The solution is to temporary set the font related language explicitly:

```
\definedfont[dejavu-sans*default]%
                                    äüöß
{\de\addfflanguage\addff{simplify}äüöß}
{\nl\addfflanguage\addff{simplify}äüöß}
```

So we can automatically switch to language specific features if we want to: äüöß aeueoesz äüöß.
Let's now move to another level of complexity: support for more than one language as in fact this example was made for Dutch in the first place, but the German outcome is a bit more visible.

```
\startluacode
fonts.handlers.otf.addfeature {
    name     = "simplify",
    type     = "multiple",
    prepend  = true,
 -- prepend  = "smcp",
    dataset  =
    {
        {
            features = {
                ["*"] = {
                    ["nld"] = true
                }
            },
            data     = {
             -- [utf.byte("ä")] = { "a" },
             -- [utf.byte("Ä")] = { "A" },
             -- [utf.byte("ü")] = { "u" },
             -- [utf.byte("Ü")] = { "U" },
             -- [utf.byte("ö")] = { "o" },
             -- [utf.byte("Ö")] = { "O" },
                [utf.byte("ĳ")] = { "i", "j" },
                [utf.byte("Ĳ")] = { "I", "J" },
                [utf.byte("æ")] = { "a", "e" },
                [utf.byte("Æ")] = { "A", "E" },
            },
        },
        {
         -- type     = "multiple", -- local values possible
            features = {
                ["*"] = {
                    ["deu"] = true
                }
            },
            data     = {
                [utf.byte("ä")] = { "a", "e" },
                [utf.byte("Ä")] = { "A", "E" },
                [utf.byte("ü")] = { "u", "e" },
                [utf.byte("Ü")] = { "U", "E" },
                [utf.byte("ö")] = { "o", "e" },
                [utf.byte("Ö")] = { "O", "E" },
                [utf.byte("ß")] = { "s", "z" },
                [utf.byte("")] = { "S", "Z" },
            },
        }
    }
}
\stopluacode
```

For this we use the following example:

```
\definedfont[dejavu-sans*default,simplify]%
                    äüöß ĳæ
{\de\addfflanguage äüöß ĳæ}
{\nl\addfflanguage äüöß ĳæ}
```

Because the Dutch is hard to check we use an æ replacement too and commented the similarities with German: äüöß ijæ aeueoesz ijæ äüöß ijae. But still we're not done, say that we want smallcaps too:

```
\definefontfeature[alwayssmcp][smcp=always]%
\definedfont[dejavu-sans*default,simplify,alwayssmcp]%
                    äüöß ĳæ
{\de\addfflanguage äüöß ĳæ}
{\nl\addfflanguage äüöß ĳæ}
```

This comes out as: äüöß ijæ aeueoesz ijæ äüöß ijae.
The reason for specifying `smcp` as `always` is that otherwise we get language specific smallcaps while often they are not bound to a language but to the defaults. The good news is that we can do this automatically:

```
\setupfonts[language=auto]%
\definefontfeature[alwayssmcp][smcp=always]%
\definedfont[dejavu-sans*default,simplify,alwayssmcp]%
     äüöß ĳæ
{\de äüöß ĳæ}
{\nl äüöß ĳæ}
```

But be aware that this applies to all situations. Here we get: äüöß ijæ aeueoesz ijæ äüöß ijae.

15.6 Syntax summary

In the examples we have seen several ways to define features. One of the differences is that you either set a `data` field directly, or that you specify a dataset. The fields in a dataset entry overload the ones given at the top level or when not set the top level value will be taken. There is a bit of (downward compatibility) tolerance built in, but best not depend on that.

```
fonts.handlers.otf.addfeature {
  name     = "demo",
  features = {
    [<script>] = {
        [<language>] = true
    }
  },
  prepend  = true | featurename | position,
  dataset  = {
     {   type = "substitution",
         data = {
             [<char|code>] = <char|code>,
         }
     },
     {   type = "alternate",
         data = {
             [<char|code>] = { <char|code>, <char|code>, ... },
         }
     },
     {   type = "multiple",
         data = {
             [<char|code>] = { <char|code>, <char|code>, ... },
         }
     },
     {   type = "ligature",
         data = {
             [<char|code>] = { <char|code>, <char|code>, ... },
         }
     },
     {   type = "kern",
         data = {
             [<char|code>] = { [<char|code>] = <value> },
         }
     },
     {   type = "pair",
         data = {
             [<char|code>] = { [<char|code>] = {
                 false | { <value>, <value>, <value>, <value> },
                 false | { <value>, <value>, <value>, <value> }
             }
         }
     },
     {
         type    = "chainsubstitution",
         lookups = {
             {
                 type = <typename>,
```

```
                    data = <mapping>,
                },
            },
            data = {
                rules = {
                    {
                        before  = { { [<char|code>], ... } },
                        current = { { [<char|code>], ... } },
                        after   = { { [<char|code>], ... } },
                        lookups = { <index>, ... },
                    },
                },
            },
        },
    },
}
```

15.7 Extra characters

You can add virtual characters to fonts. Here we give an example that is derived from an example posted on the mailing list. By default, when we hyphenated a word, we get this:

av-
ery-
long-
word

The default character that is appended at the end and beginning of a line can be specified as follows:

```
\setuplanguage
  [en]
  [righthyphenchar=45,
   lefthyphenchar=45]
```

So now we get:

av-
-ery-
-long-
-word

Say that we want a different signal, for instance some rule. Here is how that can be done:

```
\startluacode

    local privateslots = fonts.constructors.privateslots

    local function addspecialhyphen(tfmdata)
        local exheight = tfmdata.parameters.xheight
        local emwidth  = tfmdata.parameters.quad
        local width    = emwidth  /  4
        local height   = exheight / 10
        local depth    = exheight /  2
        local offset   = emwidth  /  6
        tfmdata.characters[privateslots.righthyphenchar] = {
            -- no dimensions
            commands = {

                { "right", offset },

                { "push" },
                { "right", -width },
                { "down", depth },
                { "rule", height, width },
                { "pop" },

                { "right", -width/5 },
                { "down", depth + height },
                { "rule", 3*height, width/5 },

            }
        }
        tfmdata.characters[privateslots.lefthyphenchar] = {
            -- no dimensions
            commands = {

                { "right", -offset },

                { "push" },
                { "down", depth + height },
                { "rule", 3*height, width/5 },
                { "pop" },

                { "down", depth },
                { "rule", height, width },

            }
        }
    end
```

```
    fonts.constructors.features.otf.register {
        name        = "specialhyphen",
        description = "special hyphen",
        manipulators = {
            base = addspecialhyphen,
            node = addspecialhyphen,
        }
    }

\stopluacode
```

Watch the way we use private slots. You can best use a unique glyph name as these numbers are shared between fonts. With:

```
\definefontfeature
  [default]
  [default]
  [specialhyphen=yes]
\definefont
  [DemoFont]
  [Serif*default at 24pt]
\setuplanguage
  [en]
  [righthyphenchar=\getprivateglyphslot{righthyphenchar},
   lefthyphenchar=\getprivateglyphslot{lefthyphenchar}]
```

We get:

You need to keep in mind that some of these settings are global but in practice that is not a real problem. Here is how you reset:

```
\definefontfeature
  [default]
  [default]
  [specialhyphen=no]
```

```
\setuplanguage
  [en]
  [righthyphenchar=45,
   lefthyphenchar=0]
```

15.8 Goodies

The examples above extend a font in the TEX document (normally a style) but you can use a goodies file too, for instance `cambria.lfg`.

```
return {
    name = "cambria",
    version = "1.00",
    comment = "Goodies that complement cambria.",
    author = "Hans Hagen",
    copyright = "ConTeXt development team",
    extensions = {
        {
            name = "kern", -- adds to kerns
            type = "pair",
            data = {
                [0x0153] = { -- combining acute
                    [0x0301] = { -- aeligature
                        false,
                        { -500, 0, 0, 0 }
                    }
                },
            }
        }
    }
}
```

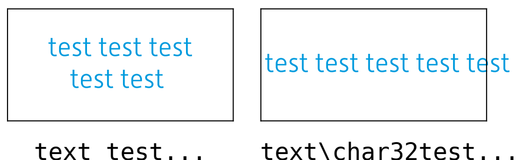
Here we use the feature name `kern` and therefore we don't have to define a specific (new) feature for it. Such a goodie is then used as follows:

```
\definefontsynonym
  [Serif] [cambria] [features=default,
                     goodies=cambria]
```

You can find such definitions in the `type-imp-*.mkiv` files.

16. Spacing

As you probably know, TEX has no space character. When the input is read, characters tagged as space are intercepted and become glue. Compare this:



`text test...` `text\char32test...`

Most fonts have a space character and you can actually use it and indeed a space character will be injected but as it is not glue, the line break algorithm will not see it as space.
Al the magic done with space characters other than the native space character (decimal 32) are at some point translated into glue.

command	UNICODE	width
`\nobreakspace` `\nbsp`	U+00A0	space
`\ideographicspace`	U+2000	quad/2
`\ideographichalffillspace`	U+2001	quad
`\twoperemspace` `\enspace`	U+2002	quad/2
`\emspace` `\quad`	U+2003	quad
`\threeperemspace`	U+2004	quad/3
`\fourperemspace`	U+2005	quad/4
`\fiveperemspace`		quad/5
`\sixperemspace`	U+2006	quad/6
`\figurespace`	U+2007	width of zero
`\punctuationspace`	U+2008	width of period
`\breakablethinspace`	U+2009	quad/8
`\hairspace`	U+200A	quad/8
`\zerowidthspace`	U+200B	0
`\zerowidthnonjoiner` `\zwnj`	U+200C	0
`\zerowidthjoiner` `\zwj`	U+200D	0
`\narrownobreakspace`	U+202F	quad/8
`\zerowidthnobreakspace`	U+FEFF	
`\optionalspace`		space when not followed by punctuation

The last one is not in UNICODE and the fifths of an emspace is not in UNICODE either. This emspace (or quad in TEX speak) is a font property. The width of the space used by ConTEXt is derived form this value. In case of a monospace fonts, the following logic is applied:

- When there is a space character, the width of that character is used.
- Otherwise, when there is an emdash present, the width if that character is used.
- Otherwise, when there is an `charwidth` property available (the average width), that value is used.

When a proportional font is used, we do as follows:

- When there is a space character, the width of that character is used.
- Otherwise, when there is an emdash present, the width of that character divided by two is used.
- Otherwise, when there is a `charwidth` property available (the average width), that value is used.

In both cases, when no value is set we use the units of the font (often 1000 or 2048). In TEX a space glue also has stretch and shrink. Here we follow the traditional TEX logic:

- The stretch is set to half the width of a space but to zero with a mono spaced font.
- The shrink is set to one third of the width of a space but to zero with a mono spaced font.

The xheight is set to the values specified by the font and when this is unset the height of the character `x` will be used but when this character is not in the font, we use two fifths of the font's units (normally the same as the emwidth). The italic angle is also taken from the font (and is of course zero for a not italic font). Most fonts have these properties set so we seldom have to fall back to a guess.

17. Ligatures

Not all fonts provide ligature control (normally related to languages), so here is a trick.

```
\blockligatures[fi,ff]
\blockligatures[fl]

\definefontfeature
  [default]
  [default]
  [blockligatures=yes]

\setupbodyfont[pagella]

...
```

This way it works globally. Of course you can also bind it to a font instance:

```
\blockligatures[fi,fl]

\definefontfeature
  [default:blockligs]
```

```
  [default]
  [blockligatures=yes]

\definefont[DemoBlockY][dejavu-serif*default:blockligs at 20pt]
\definefont[DemoBlockN][dejavu-serif*default           at 20pt]

Here we have no ligatures: {\DemoBlockY fi ff fl}, while here we
get them: {\DemoBlockN fi ff fl}. Of course it also depends on
the font.
```

Here we have no ligatures: fi ff fl, while here we get them: ﬁ ﬀ ﬂ. Of course it also depends on the font.

There is one limitation: you need to specify the blocked ligatures before a font gets defined and because we share resources it even has to happen before the first font gets loaded. So, the `\blockligatures` commands go before setting up the body font. This is no real problem because it's a hack anyway.

The next example combines several tricks:

```
\startluacode
    fonts.handlers.otf.addfeature {
        name = "kernligatures",
        type = "kern",
        data = {
            f = { i = 50, l = 50 },
        }
    }
\stopluacode

\blockligatures[u:fl:a]

\definefontfeature[default:b][default]
  [blockligatures=yes]
\definefontfeature[default:k][default]
  [blockligatures=yes,kernligatures=yes]

\showfontkerns
```

```
{\definedfont[Brill*default   @ 11pt]auflage}\par
{\definedfont[Brill*default:b @ 11pt]auflage}\par
{\definedfont[Brill*default:k @ 11pt]auflage}\par
```

auflage -0.110

auflage -0.110

auflage 0.550 -0.110

Processing fonts is complicated by the fact that a text can be hyphenated. This complicates for instance ligature building which can cross the pre, post and/or replace bounds. The current implementation does a decent job although there will always be border cases. And, figuring out what goes wrong is a pain. There are several ways to trace what happens and here's one. As mentioned, blocking only works when we haven't not yet defined a font instance, so we use a funny size here.

```
\blockligatures[u:fl:a]

\definefontfeature
  [blockligatures]
  [default]
  [blockligatures=yes]

\startotfcompositionlist
    {texgyrepagella-regular*blockligatures @ 10pt}{l2r}
    \HL
    \showotfcompositionsample{auflage}
    \showotfcompositionsample{a\discretionary{-}{}{}uflage}
    \showotfcompositionsample{au\discretionary{-}{}{}flage}
    \showotfcompositionsample{auf\discretionary{-}{}{}lage}
    \showotfcompositionsample{aufl\discretionary{-}{}{}age}
    \showotfcompositionsample{aufla\discretionary{-}{}{}ge}
    \showotfcompositionsample{auflag\discretionary{-}{}{}e}
    \HL
    \showotfcompositionsample{auflegt}
    \showotfcompositionsample{a\discretionary{-}{}{}uflegt}
    \showotfcompositionsample{au\discretionary{-}{}{}flegt}
    \showotfcompositionsample{auf\discretionary{-}{}{}legt}
    \showotfcompositionsample{aufl\discretionary{-}{}{}egt}
    \showotfcompositionsample{aufle\discretionary{-}{}{}gt}
    \showotfcompositionsample{aufleg\discretionary{-}{}{}t}
    \HL
\stopotfcompositionlist
```

auflage	au[-||]flage	au[-||]flage	auflage
a\discretionary{-}{}{}uflage	a[-||]uflage	a[-||]uflage	auflage
au\discretionary{-}{}{}flage	au[-||]flage	au[-||]flage	auflage
auf\discretionary{-}{}{}lage	auf[-||]lage	au[f-|l|fl]age	auflage
aufl\discretionary{-}{}{}age	aufl[-||]age	aufl[-||]age	auflage
aufla\discretionary{-}{}{}ge	aufla[-||]ge	aufla[-||]ge	auflage
auflag\discretionary{-}{}{}e	auflag[-||]e	auflag[-||]e	auflage
auflegt	au[-||]flegt	au[-||]flegt	auflegt
a\discretionary{-}{}{}uflegt	a[-||]uflegt	a[-||]uflegt	auflegt
au\discretionary{-}{}{}flegt	au[-||]flegt	au[-||]flegt	auflegt
auf\discretionary{-}{}{}legt	auf[-||]legt	au[f-|l|fl]egt	auflegt
aufl\discretionary{-}{}{}egt	aufl[-||]egt	aufl[-||]egt	auflegt
aufle\discretionary{-}{}{}gt	aufle[-||]gt	aufle[-||]gt	auflegt
aufleg\discretionary{-}{}{}t	aufleg[-||]t	aufleg[-||]t	auflegt

Here is another example. This one demonstrates that ligatures can force collapsing of discretionaries.

```
\startotfcompositionlist{Serif*default @ 11pt}{l2r}
    \HL
    \showotfcompositionsample{effe}
    \showotfcompositionsample{efficient}
    \HL
\stopotfcompositionlist
```

effe	effe	effe	effe
efficient	ef[-||]fi[-||]cient	e[f-|fi|ffi][-||]cient	efficient