Fonts out of ConTeXt

explaining luatex and mkiv

Hans Hagen

PRAGMA ADE

Contents

Contents

Contents

Introduction

You sit in a cave and wonder how to keep track of your winter stock. While playing with some burned wood you end up with vertical strokes on the wall representing how much you have in store.

You walk through the woods and wonder how to find your way back. Suddenly it strikes you that you can put markers on trees. Years from that moment the whole forest is marked with routes. Different symbols carry different meanings.

The next thing you want to do is to carry around information and pass it onto following generations. So, you turn those symbols into shapes that make up the scripts that can be used to express your languages in.

For ages scripts have evolved and the rendering of them on stone or wood and later paper has resulted in a multitude of coherent collections of so called glyphs. Manual labour turned into (semi) automated mass production and once that took off, developments went fast. But the quality was still somewhat dubious, especially when for instance specialized scripts like math had to be dealt with.

Some 30 years ago Don Knuth wrote a book, and in the process invented the TeX typesetting system, the graphical language METAFONT and a bunch of fonts. He made it open and free of charge. He was well aware that the new ideas were built on older ones that had evolved from common sense: how to keep track of things on paper.

It is no surprise that an active community formed around these goodies. First of all the system has no strings attached: the licence is generous and there are no patents involved. There is also a network of user groups that takes care of coordinated updates to the whole machinery. Of course it helps that it all relates to Don Knuth.

Since TeX showed up several open and closed source typesetting systems have surfaced and only some of them survived. Also regular word processing has become more clever and still become better. The TeX typesetting system also moved on. Some of its ideas have been used in other programs and some of the ideas of other programs made their way into TeX. However, its main property is still there: you can tweak and tune it to your needs and are not hampered by too many limitations.

Don Knuth had this chicken or egg problem: once you can typeset a source you need fonts but you can only make fonts if you can use them in a typesetting program. As a result TeX came with its own fonts and it has special ways to deal with them. Given the limitations of that time TeX puts some limitations on fonts and also expects them to have certain properties, something that is most noticeable in math fonts.

Rather soon from the start it has been possible to use third party fonts in TeX, for instance Type1. As TeX only needs some information about the shapes, it was the backend that integrated the font resources in the final document. One of its descendants, pdfTeX, had

this backend built in and could do some more clever things with fonts in the typesetting process, like protrusion and expansion. The integration of front- and backend made live much easier. Another descendant, X͟TEX made it possible to move on to the often large OpenType fonts. On the one hand this made live even more easy but at the other end it introduced users to the characteristics of such fonts and making the right choices, i.e. not fall in the trap of too fancy font usage.

In this manual we will look at fonts from the perspective of yet another descendant, LuaTEX. It inherits the font technology from traditional TEX, but also extends it so that we can deal with modern font technologies. Of course it offers much more, but in practice much relates to fonts one way or the other.

Of course this exploration will be from the perspective of the ConTEXt macro package but this is not a manual about how to use fonts in ConTEXt as we have another manual for that. Much of what we say here applies to the generic font code as well, although some more advanced control is ConTEXt specific. There is nothing real new here, and it all evolved from common sense and dealing with TEX for many years. The perspective is mostly that of being a user myself so don't complain too loudly if things look complicated and unclear.

There is some overlap between the chapters. This is because each chapter is written from another perspective and this document quite certainly will not be read as a whole but more by looking at examples.

This document will probably have an 'still under construction' state for a long time. The functionality discussed here will stay and more might show up. Of course there are errors, and they're all mine.

Hans Hagen
PRAGMA ADE, Hasselt NL
Summer 2011 – Spring 2016

1 Font formats

1.1 Introduction

In this chapter the font formats as we know them will be introduced. The descriptions will be rather general but more details can be found in the appendix. Although in MkIV we do support all these types eventually the focus will be on OpenType fonts but it does not hurt to see where we are coming from.

1.2 Glyphs

A typeset text is mostly a sequence of characters turned into glyphs. We talk of characters when you input the text, but the visualization involves glyphs. When you copy a part of the screen in an open pdf document or html page back to your editor you end up with characters again. In case you wonder why we make this distinction between these two states we give an example.

affiliation *affiliation*

upright   italic

Figure 1.1   From characters to glyphs.

We see here that the shape of the `a` is different for an upright serif and an italic. We also see that in `ffi` there is no dot on the `i`. The first case is just a stylistic one but the second one, called a ligature, is actually one shape. The 11 characters are converted into 9 glyphs. Hopefully the final document format carries some extra information about this transformation so that a cut and paste will work out well. In pdf files this is normally the case. In this document we will not be too picky about the distinction as in most cases the glyph is rather related to the character as one knows it.

So, a font contains glyphs and it also carries some information about replacements. In addition to that there needs to be at least some information about the dimensions of them. Actually, a typesetting engine does not have to know anything about the actual shape at all.

a b g l q . ; ? ffi

Figure 1.2   The boundingbox
of some normal glyphs.

$$a \; b \; g \; l \; q \; . \; ; \; ? \; ffi$$

Figure 1.3 The boundingbox
of some italic glyphs.

The rectangles around the shapes figure 1.2 and figure 1.3 are called boundingbox. The dashed line reflects the baseline where they eventually are aligned onto next to each other. The amount above the baseline is called height, and below is called depth. The piece of the shape above the baseline is the ascender and the bit below the descender. The width of the bounding box is not by definition the width of the glyph. In Type1 and OpenType fonts each shape has a so called advance width and that is the one that will be used.

Very often glyphs get very small spaces inserted horizontally.

Very often glyphs get very small spaces inserted horizontally.

Very often glyphs get very small spaces inserted horizontally.

Very often glyphs get very small spaces inserted horizontally.

Figure 1.4 Kerning in Latin Roman, Cambria, Pagella and Dejavu.

Another traditional property of a font is kerning. In figure 1.4 you see this in action. These examples demonstrate that not all fonts need (or provide) the same kerns (in points).

So, as a start, we have now met a couple of properties of a font. They can be summarized as follows:

mapping to glyphs	:	characters are represented by a shapes that have recognizable properties so that readers know what they mean
ligature building	:	a sequence of characters gets mapped onto one glyph
dimensions	:	each glyph has a width, height and depth
inter-glyph kerning	:	optionally a bit of positive or negative space has to be inserted between glyphs

Regular font kerning is hardly noticeable and improves the overall look of the page. Typesetting applications sometimes are capable of inserting additional spaces between shapes. This more excessive kerning is not that much related to the font and is used for special purposes, like making a snippet of text stand out. In ConTEXt this kind of kerning is available but it is a font independent feature. Keep in mind that when applying that kind of rather visible kerning you'd better not have ligatures and fancy replacements enabled as ConTEXt already tries to deal with that as good as possible.

1.3 The basic process

In TEX a font is an abstraction: the engine only needs to know about the mapping from characters to glyphs, what the width, height and depth is, what sequences need to be translated into ligatures and when kerning has to be applied. If for the moment we forget about math, these are all the properties that matter and this is what the TEX font metric files that we see in the next section provide.

Because one of the principles behind LuaTEX is that the core engine (the binary) stays small and that new functionality is provided in Lua code, the font subsystem largely looks like it always has been. As users will normally use a macro package most of the loading will be hidden from the user. It is however good to give a quick overview of how for instance pdfTEX deals with fonts using traditional metric files.

Figure 1.5 Several translation steps in a traditonal TEX flow.

The input (bytes) gets translated into characters by the input parser. Normally this is a one-to-one translation but there are examples of some translation taking place. You can for instance make characters active and give them a meaning. So, the eight bit representation of an editors code page ë can become something else internally, for instance a regular e with an ¨ overlayed. It can also become another character, which in the code page would be shown as á but the user will not know this as by then this byte is already tokenized. Another example is multibyte translation, for instance utf sequences can get remapped to something that is known internally as being a character of some kind. The LuaTEX engine expects utf so a macro package has to make sure that translation to this encoding happens beforehand, for instance using a callback that intercepts the input from file.[1]

So, the input character (sequence) becomes tokens representing a character. From these tokens TEX will start building a (linked) node list where each character becomes a node. In this node there is a reference to the current font. If you know TEX you will understand that a list can have more than characters: there can be skips, kerns, rules, references to images, boxes, etc.

At some point TEX will handle this list over to a routine that will turn them into something that resembles a paragraph or otherwise snippet of text. In that stage hyphenation kicks in, ligatures get built and kerning is added. Character references become glyph indices. This list can finally be broken into lines.

It is no secret that TEX can box and unbox material and that after unboxing some new formatting has to happen. The traditional engine has some optimizations that demand

---

[1] In ConTEXt we talk of input regimes and these can be mixed, although in practice most users will stick to utf and never use regimes.

a partial reconstruction of the original list but in LuaTeX we removed this kind of optimization so there the process is somewhat simpler. We will see more of that later.

When TeX ships out a page, the backend will load the real font data and merge that into the final output. It will now take the glyph index and build the right data structures and references to the real font. As a font gets subset only the used glyphs end up in the final output.

There is one tricky aspect involved here: re-encoding. In so called map files one can map a specific metric filename onto a real font name. One can also specify an encoding vector that tells what a given index really refers to. This makes it possible to use fonts that have more than 256 glyphs and refer to any of them. This is also the trick that makes it possible to use TrueType fonts in pdfTeX: the backend code filters the right glyphs from the font, remapping TeX's glyph indices onto real entries in the font happens via the encoding vector. In figure 1.6 we show a possible route for input byte 68.

Figure 1.6   From bytes to indices.

As LuaTeX carries much of the bagage of older engines, you can still do it this way but in ConTeXt MkIV we have made our live much simpler: we use unicode as much as possible. This means that we effectively have removed two steps (see figure 1.7).

Figure 1.7   Simplified mapping in LuaTeX.

There is of course still some work to do for the backend, like subsetting, but the nasty dependency on the input encoding, font encoding (that itself relates to hyphenation) and backend re-encoding is gone. But keep in mind that the internal data structure of the font is still quite traditional.

Before we move on to font formats I like to point out that there is no space in TeX. Spaces in the input are converted into glue, either or not with some stretch and/or shrink. This also means that accessing character 32 in traditional TeX will not end up as space in the output.

1.4 TeX metrics

Traditional font metrics are packaged in a binary format. Due to the limitations of that time a font has at most 256 characters. In books dedicated to TeX you will often find tables that show what glyphs are in a font, so we will not repeat that here as after all we got rid of that limitation in LuaTeX.

Because 256 is not that much, especially when you mix many scripts and need lots of symbols from the same font, there are quite some encodings used in traditional TEX, like texnansi, ec and qx. When you use LuaTEX exclusively you can do with way less font files. This is easier for users, especially because most of those files were never used anyway. It's interesting to notice that some of the encodings contain symbols that are never used or used only once in a document, like the copyright or registered symbols. They are often accessed by symbolic names and therefore easily could have been omitted and collected in a dedicated symbol font thereby freeing slots for more useful characters anyway. The lack of coverage is probably one of the reasons why new encodings kept popping up. In the next table you see how many files are involved in Latin Modern which comes in a couple of design sizes.[2]

font format	type	# files	size in bytes	ConTEXt
type 1	tfm	380	3.841.708	
	afm	25	2.697.583	
	pfb	92	9.193.082	
	enc	15	37.605	
	map	9	42.040	
		521	15.812.018	mkii
opentype	otf	73	8.224.100	mkiv

A tfm file can contain so called italic corrections. This is an additional kern that can be added after a character in order to get better spacing between an italic shape and an upright one. As this is manual work, it's a not that advanced mechanism, but in addition to width, height, depth, kerns and ligatures it is nevertheless a useful piece of information. But, it's a rather TEX specific quantity.

Since TEX showed up many fonts have been added. In addition support for commercial fonts was provided. In fact, for that to happen, one only needs accompanying metric files for TEX itself and map files and encoding vectors for the backend. Because a metric file also has some general information, like spacing (including stretch and shrink), the ex-height and em-width, this means that sometimes guesses must be made when the original font does not come with such parameters.

At some point virtual fonts were introduced. In a virtual font a tfm file has an accompanying vf file. In that file each glyph has a specification that tells where to find the real glyph. It is even possible to construct glyphs from other glyphs. In traditional TEX this only concerns the backend, which in pdfTEX is built in. In LuaTEX this mechanism is integrated into the frontend which means that users can construct such virtual fonts themselves. We will see more of that later, but for now it's enough to know that when we talk about the representation of font (the tfm table) in LuaTEX, this includes virtual functionality.

---

[2] The original Computer Modern fonts have METAFONT source files and (runtime) generated bitmap files in whatever resolutions are needed for previewing and printing. The Type1 follow-up came in several sets, organized by language support. The Latin Modern fonts have a few more weights and variants than Computer Modern.

An important limitation of tfm files cq. traditional TeX is that the number of depths and heights is limited to 16 each. Although this results in somewhat inaccurate dimensions in practice this gets unnoticed, if only because many designs have some consistency in this. On the other hand, it is a limitation when we start thinking of accents or even multiple accents which lead to many more distinctive heights and depths.

Concerning ligatures we can remark that there are quite some substitutions possible although in practice only the multiple to one replacement has been used.

Some fonts that are used in TeX started out as bitmaps but rather soon Type1 outline fonts became the fashion. These are supported using the map files that we will discuss later. First we look into Type1 fonts.

1.5 Type1

A.5
A.4
A.3

For a long time Type1 fonts have dominated the scene. These are PostScript fonts that can have more that 256 glyphs in the file that defines the shapes, but only 256 of them can be used at one time. Of course there can be multiple subsets active in one document.

In traditional TeX a Type1 font is used by making a tfm file from a so called Adobe metric file (afm) that come with such a font. There are several tool chains for doing this and ConTeXt MkII ships with one that can be of help when you need to support commercial fonts. Projects like the Latin Modern Fonts and TeX Gyre have normalized a whole lot of fonts that came in several more or less complete encodings into a consistent package of Type1 fonts. This already simplified live a lot but still users had to choose a suitable input and font encoding for their language and/or script. As TeX only cares about metrics and not about the rendering, it doesn't consider Type1 fonts as something special. Also, as TeX and PostScript were developed about the same time support for Type1 fonts is rather present in TeX distributions.

You can still follow this route but for ConTeXt MkIV this is no longer the recommended way because there we have changed the whole subsystem to use Unicode. As a result we no longer use tfm files derived from afm files, but directly interpret the afm data. This not only removes the 256 limitation, but also brings more resolution in height and depth as we no longer have at most 16 alternatives. There can also be more kerns. Of course we need some heuristics to determine for instance the spacing but that is not different from former times.

Because most TeX users don't use commercial fonts, they will not notice that ConTeXt MkIV treats Type1 fonts this way. One reason is that the free fonts also come as wide fonts in OpenType format and whenever possible ConTeXt prefers OpenType over Type1 over tfm.

In the beginning LuaTeX only could load a tfm file, which is why loading afm files is implemented in Lua. Later, when the OpenType loaded was added, loading pfb and afm files also became possible but it's slower and we see no reason to rewrite the current

code in ConTEXt. We also do a couple of extra things when loading such a file. As more Type1 fonts move on to OpenType we don't expect that much usage anyway.

1.6 OpenType

When an engine can deal with Unicode directly it also means that internally it uses pretty large numbers for storing characters and glyph indices. The first TEX descendent that went wide was Omega, later replaced by Aleph. However, this engine never took off and still used its own extended tfm format: ofm. In fact, as LuaTEX uses some of the Aleph code, it can also use these extended metric files but I don't think that there are any useful fonts around so we can forget about this.

We use the term OpenType for a couple of font formats that share the same principles: OpenType (otf), TrueType (ttf) and TrueType containers (ttc). The LuaTEX font reader presents them in a similar format. In the case of a TrueType container, one does not load the whole font but selects an instance from it. Internally an OpenType font can have the glyphs organized in subfonts.

The first TEX descendent to really go wide from front to back is X∃TEX. This engine can use OpenType fonts directly and for a whole category of users this opened up a new world. Hoever, it is still mostly a traditional engine. The transition from characters to glyphs is accomplished by external libraries, while in LuaTEX we code in Lua. This has the disadvantage that it is slower (although that depends on the job) but the advantage is that we have much more control and can extend the font handler as we like.

An OpenType font is much more complex than a Type1 one. Unless it is a quick and dirty converted existing font, it will have more glyphs to start with. Quite likely it will have kerns and ligatures too and of course there are dimensions. However, there is no concept of a depth and height. These need to be deduced from the bounding box instead. There is an advance width. This means that we can start right away using such fonts if we map those properties onto the tfm table that LuaTEX expects.

But there is more, take ligatures. In a traditional font the sequence ffi always becomes a ligature, given that the font has such a glyph. In LuaTEX there is a way to disable this mechanism, which is sometimes handy when dealing with mono-spaced fonts in verbatim. It's pretty hard to disable that. For instance one option is to insert kerns manually. In an OpenType font ligatures are collected in a so called feature. There can be many such features and even kerning is a feature. Other examples are old style numerals, fractions, superiors, inferiors, historic ligatures and stylistic alternates.
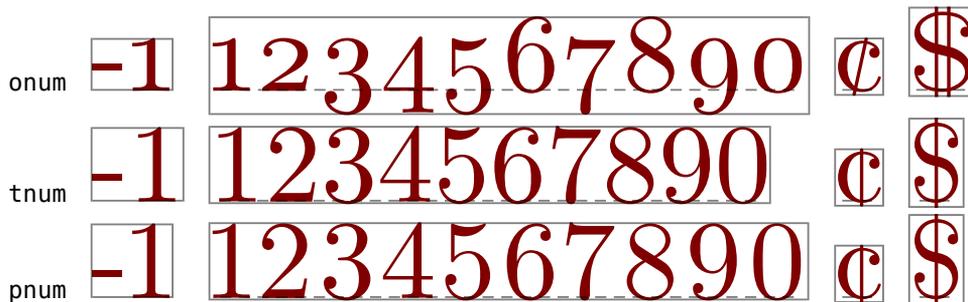
To this all you need to add that features operate in two dimensions: languages and scripts. This means that when ligatures are enabled for Dutch the `ij` sequence becomes a single glyph but for German it gets mapped onto two glyphs. And, to make it even more complex, a substitution can depend on circumstances, which means that for Dutch `fijn` becomes f `ij` n but `fiets` becomes fi ets. It will be no surprise that not all OpenType fonts come with a complete and rich repertoire of rules. To make things worse, there can be rules that turn `1/2` into one glyph, or transfer the numbers into superior and inferior alternatives, but leaves us with an unacceptable rendered `1/a`, given that the `frac` features is enabled. It looks like features like this are to be applied to a manually selected range of characters.

The fact that an OpenType font can contain many features and rules to apply them makes it possible to typeset scripts like Arabic. And this is where it gets vague. A generic OpenType sub-engine can do clever things using these rules, but if you read the specification for some scripts additional intelligence has to be provided by the typesetting engine.

While users no longer have to care about encodings, map files and back-end issues, they do have to carry knowledge about the possibilities and limitations of features. Even worse, he or she needs to be aware that fonts can have bugs. Also, as font vendors have no tradition of providing updates this is something that we might need to take care of ourselves by tweaking the engine.

One of the problems with the transition from Type1 to OpenType is that font designers can take an existing design and start from that basic repertoire of shapes. If such a design had oldstyle figures only, there is a good chance that this will be the case in the OpenType variant too. However, such a default interferes with the fact that the `onum` feature is one that we explicitly have to enable. This means that writing a generic style where a font is later plugged in becomes somewhat messy if it assumes that features need to be turned on.

TeX users expect more control, which means that in practice just an OpenType engine is not enough, but for the average font the TeX model using the traditional approach still is quite acceptable. After all, not all users use complex scripts or need advanced features. And, in practice most readers don't notice the difference anyway.

1.7 Lua

As mentioned support for virtual fonts is built into LuaTEX and loading the so called vf
files happens when needed. However, that concerns traditional fonts that we already
covered. In ConTEXt we do use the virtual font mechanism for creating missing glyphs
out of existing ones or add fallbacks when this is not possible. But this is not related to
some kind of font format.

In 2010 and 2011 the first public OpenType math fonts showed up that replace their
Type1 originals. In ConTEXt we already went forward and created virtual Unicode fonts
out of traditional fonts. Of course eventually the defaults will change to the OpenType
alternatives. The specification for such a virtual font is given in Lua tables and therefore
you can consider Lua to be a font format as well. In ConTEXt such fonts can be defined
in so called goodies files. As we use these files for much more tuning, we come back to
that in a later chapter. In a virtual font you can mix real Type1 fonts and real OpenType
fonts using whatever metrics suit best.

An extreme example is the virtual Unicode Punk font. This font is defined in the Me-
taPost language (derived from Don Knuths METAFONT sources) where each glyph is one
graphic. Normally we get PostScript, but in LuaTEX we can also get output in a compa-
rable Lua table. That output is converted to pdf literals that become part of the virtual
font definitions and these eventually end up in the pdf page stream. So, at the TEX end
we have regular (virtual) characters and all TEX needs is their dimensions, but in the pdf
each glyph is shown using drawing operations. Of course the now available OpenType
variant is more efficient, but it demonstrates the possibilities.

1.8 Files

We summarize these formats in the following table where we explain what the file suf-
fixes stand for:

tfm	This is the traditional TEX font metric file format and it reflects the internal quantities that TEX uses. The internal data structures (in LuaTEX) are an extension of the tfm format.
vf	This file contains information about how to construct and where to find virtual glyphs and is meant for the backend. With LuaTEX this format gets more known.
pk	This is the bitmap format used for the first generation of TEX fonts but the typesetter never deals with them. Bitmap files are more or less obsolete.
ofm	This is the Omega variant of the tfm files that caters for larger fonts.
ovf	This is the Omega variant of the vf.
pfb	In this file we find the glyph data (outlines) and some basic information about the font, like name-to-index mappings. A differently byte-encoded variant of this format is pfa.

afm   This file accompanies the `pfb` file and provides additional metrics, kerns and infor-
mation about ligatures. A binary variant of this is the pfa format. For MS Windows
there is a variant that has the `pfm` suffix.

map   The backend will consult this file for mapping metric file names onto real font
names.

enc   The backend will include (and use) this encoding vector to map internal indices to
font indices using glyph names, if needed.

otf   This binary format describes not only the font in terms of metrics, features and
properties but also contains the shapes.

ttf   This is the Microsoft variant of OpenType.

ttc   This is the Microsoft container format that combines multiple fonts in one.

fea   A (FontForge) feature definition file. Such a file can be loaded and applied to a
font. This is no longer supported in ConTEXt as we have other means to achieve
the same goals.

cid   A glyph index (name) to Unicode mapping file that is referenced from an OpenType
font and is shared between fonts.

lfg   These are ConTEXt specific Lua font goodie files providing additional information.

If you look at how files are organized in a TEX distribution, you will notice that these
files all get their own place. Therefore adding a Type1 font to the distribution is not that
trivial if you want to avoid clashes. Also, files are simply not found when they are not in
the right spot. Just to mention a few paths:

```
<root>/fonts/tfm/vendor/typeface
<root>/fonts/vf/vendor/typeface
<root>/fonts/type1/vendor/typeface
<root>/fonts/truetype/vendor/typeface
<root>/fonts/opentype/vendor/typeface
<root>/fonts/fea
<root>/fonts/cid
<root>/fonts/dvips/enc
<root>/fonts/dvips/map
```

There can be multiple roots and the right locations are specified in a configuration file.
Currently all engines can use the dvips encoding and map files, so luckily we don't
need to duplicate this. For some reason TrueType and OpenType fonts have different
locations and you need to be aware of the fact that some fonts come in both formats
(just to confuse users) so you might end up with conflicts.

In ConTEXt we try to make live somewhat easier by also supporting a simple path struc-
ture:

```
<root>/fonts/data/vendor/typeface
```

This way files are kept together and installing commercial fonts is less complex and error prone. Also, in practice we only have one set of files now: one of the other OpenType formats.

If you want to see the difference between a traditional (pdfTEX or XƎTEX plus ConTEXt MkII) setup or a modern one (LuaTEX with ConTEXt MkIV) you can install the ConTEXt suite (formerly known as minimals). If you explicitly choose for a LuaTEX only setup, you will notice that far less files get installed.

1.9 Text

This is not an in-depth explanation of how to define and load fonts in ConTEXt. First of all this is covered in other manuals, but more important is that we assume that the reader is already familiar with the way ConTEXt deals with fonts. Therefore we limit ourselves to some remarks and expand on this a bit in later chapters.

The font subsystem has evolved over years and when you look at the low level code you will probably find it complex. This is true, although in some aspects it is not as complex as in MkII where we also had to deal with encodings due to the eight bit limitations. In fact, setting up fonts is easier due the fact that we have less files to deal with.

The main properties of a (modern) font subsystem for typesetting text are the following:

1. We need to be able to switch the look and feel efficiently and consistently, for instance going from regular to bold or italic. So, when we load a font family we not only load one file, but often at least four: regular, bold, italic (oblique) and bolditalic (boldoblique).

2. When we change the size we also need to make sure that these related sets are changed accordingly. You really want the bold shapes to scale along with the regular ones.

3. Shapes are organized in serif, sans serif, mono spaced and math and for proper working of a typesetter that has math all over you need always need the math. Again, when you change size, all these shapes need to scale in sync.

4. In one document several families can be combined so the subsystem should make it possible to switch from one to the other without too much overhead.

5. Because section heads and other structural elements have their own sizes there has to be a consistent way to deal with that. It should also be possible to specify exceptions for them.

In the next chapters we will cover some details, for instance font features. You can actually control these when setting up a body font, simply by redefining the `default` feature set, but not all features are dealt with this way. So let's continue the demands put on a font subsystem.

6. Sometimes inter-character kerning is needed. In ConTEXt this is not a property of a font because glyphs can be mixed with basically anything. This kind of features is applied independent of a font.

7. The same is true for casing (like uppercasing and such) which is not related to a font but applied to a selected (or marked) piece of the input stream.

8. Using so called "small caps" or "old style" numerals or ... can be dealt with by setting the default features but often these are applied selectively. As these are applied using the information in a font they do belong to the font subsystem but in practice they can be seen as independent (assuming that the font supports them at all).

9. Protrusion (into margins) and expansion (to improve whitespace) are applied to the font at load time because the engine needs to know about them. But they two can selectively be turned on and off. They are more related to line break handling than font defining.

10. Slanting (to fake oblique) and expanding (to fake bold) are regular features but are applied to the font because the engine needs to know about them. They permanently influence the shape.

We will discuss these in this manual too. What we will not discuss in depth is spacing, even when it depends on the (main body) font size. These use properties of fonts (like the ex-height or em-width and maybe the width of the space, but normally they are controlled by the spacing subsystem. We will however mention some rather specific possibilities:

11. The ConTEXt font subsystem provides ways to combine multiple fonts into one.

12. You can construct artificial fonts, using existing fonts or MetaPost graphics.

13. Fonts can be fixed (dimensions) and completed (for instance accented characters) when loading/

14. There are extensive tracing options, not only for applied features but also for loading, checking etc. There is a set of styles that can be used to study fonts.

Sometimes users ask for very special trickery and it no surprise then that some of that is now widely know (or even discussed in detail). When we get notice of that we can mention it in this manual.

So how does this all relate to font formats? We mentioned that when loading we basically load some four files per family (and more if we use specific fonts for titling). These files just provide the data: metric information, shapes and ways to remap characters (or sequences) into glyphs, either of not positioned relative to each other. In traditional TEX only dimensions, kerns and ligatures mattered, but in nowadays we also deal with specific OpenType features. But still, as you can deduce from the above, this is only part of the story. You need a complete and properly integrated system. It is no big deal to

set up some environment that uses font files to achieve some typesetting goal, but to provide users with some consistent and extensible system is a bit more work.

There are basically three font formats: good old bitmaps, Type1 and OpenType. All need to be supported and expectations are that we also support their features. But is should be noticed that whatever font you use, the quality of the outcome depends on what information the font can provide. We can improve processing but are often stuck with the font. There are many thousands of fonts out there and we need to be able to use them all.

1.10 Math

In the previous section we already mentioned math fonts. The fonts are just one aspect of typesetting math and math fonts are special in the sense that they have to provide the relevant information. For instance a parenthesis comes in several sizes and at some point turns in a symbol made out of pieces (like a top curve, middle lines and bottom curve) that overlap. The user never sees such details. In fact, there are ot that many math fonts and these are already set up so there is not much to mess up here. Nevertheless we mention:

1. Math fonts are loaded in three sizes: text, script and scriptscript. The optimal relative sizes ar defined in the font.

2. There are direction aware math fonts and we support this in ConTEXt.

3. Bold math is in fact a bolder version of a regular math font (that can have bold symbols too). Again this is supported.

The way math is dealt with in ConTEXt is different from the way it is done traditionally. Already when we started with MkIV we moved to Unicode and the setup at the font level is kept simple by delegating some of the work to the Lua end. We will see some of the mentioned aspects in more detail later.

Because of it's complexity and because in a math text there can be many times activation of math fonts (and related settings) quite some effort has been put in making it efficient. But you need to keep in mind that when we discuss math related topics later on, this is hardly of concern. Math fonts are loaded only once so manipulating them a bit has no penalty. And using them later on is hardly related to the font subsystem.

Concerning formats we can notice that traditional TEX comes with math fonts that have properties that the engine can use. Because there were not many math fonts, this was no problem. The OpenType math fonts however are also used in other applications and therefore are a bit more generic.[3] For this we not only had to adapt the math engine in LuaTEX (although we kept that to the minimum) but we also had to think different about loading them. In later chapters we will see that in the transition to Unicode math fonts

---

[3] Their internals are now defined in the OpenType specification.

we implemented a mechanism for combining Type1 fonts into virtual Unicode fonts. We did that because it made no sense to keep an old and new loader alongside.

There will not be thousands of math fonts flying around. A few dozen is already a lot and the developers of macro packages can set them up for the users. So, in practice there is not much that a user needs to know about math font formats.

1.11 Caching

Because fonts can be large and because we use Lua tables to describe them a bit of effort has been put into managing them efficiently. Once converted to the representation that we need they get cached. You can peek into the cache which is someplace on your system (depending on the setup):

`fonts/afm`	type one fonts, converted from `afm` and `pfb` files
`fonts/data`	font name databases
`fonts/mp`	fonts created using MetaPost
`fonts/otf`	open type fonts, converted from `ttf`, `otf`, `ttc` and `ttx` files loaded using the FontForge loader
`fonts/otl`	open type fonts, converted from `ttf`, `otf`, `ttc` and `ttx` files loaded using the ConTEXt Lua loader
`fonts/shapes`	outlines of fonts (for instance for use in MetaFun)

There can be three types of files there. The `tma` files are just Lua tables and they can be large. These files can be compiled to bytecode where `tmc` is for stock LuaTEX and `tmb` for LuajitTEX. The `tma` files are optimized for space and memory (aka: packed) but you can expand them with `mtxrun --script font`.

Fonts in the cache are automatically updated when you install new versions of a font or when the ConTEXt font loader has been updated.

2 Modes

2.1 Introduction

We use the term modes for classifying the several ways characters are turned into glyphs. When a font is defined, a set of features can be associated and one of them is the mode.

none   Characters are just mapped onto glyphs and no substitution or positioning takes place.
base   The routines built into the engine are used. For many Latin fonts this is a rather useable and efficient method.
node   Here alternative routines written in Lua are used. This mode is needed for more complex scripts as well as more advanced features that demand some analysis.
auto   This mode will determine the most suitable mode for the given feature set.

When we talk about features, we refer to more than only features provided by fonts as ConTEXt adds some of its own. In the following section each of these modes is discussed. Before we do so a short introduction to font tables that we use is given.

2.2 The font table

The internal representation of a font in ConTEXt is such that we can conveniently access data that is needed in the mentioned modes. When a font is used for the first time, or when it has changed, it is read in its most raw form. After some cleanup and normalization the font gets cached when it is a Type1 or OpenType font. This is done in a rather efficient way. A next time the cached copy is used.

The normalized table is shared among instances of a font. This means that when a font is used at a different scale, or when a different feature set is used, the font gets loaded only once and its data is shared when possible. In figure 2.1 we have visualized the process. Say that you ask for font `whatever` at `12pt` using featureset `smallcaps`. In low level code this boils down to:

```
\font\MySmallCaps=whatever*smallcaps at 12pt
```

In ConTEXt we have overloaded the font loader so Lua code takes care of the loading. Basically there is a function hooked into LuaTEX's font definer (the `\font` primitive) that returns a table and from that on LuaTEX will create its internal representation that is identified by a number, the so called font id. So, in fact the `\Whatever` command is a reference to a font id, a positive number. When this font is already loaded, ConTEXt will reuse the id and pas that one.
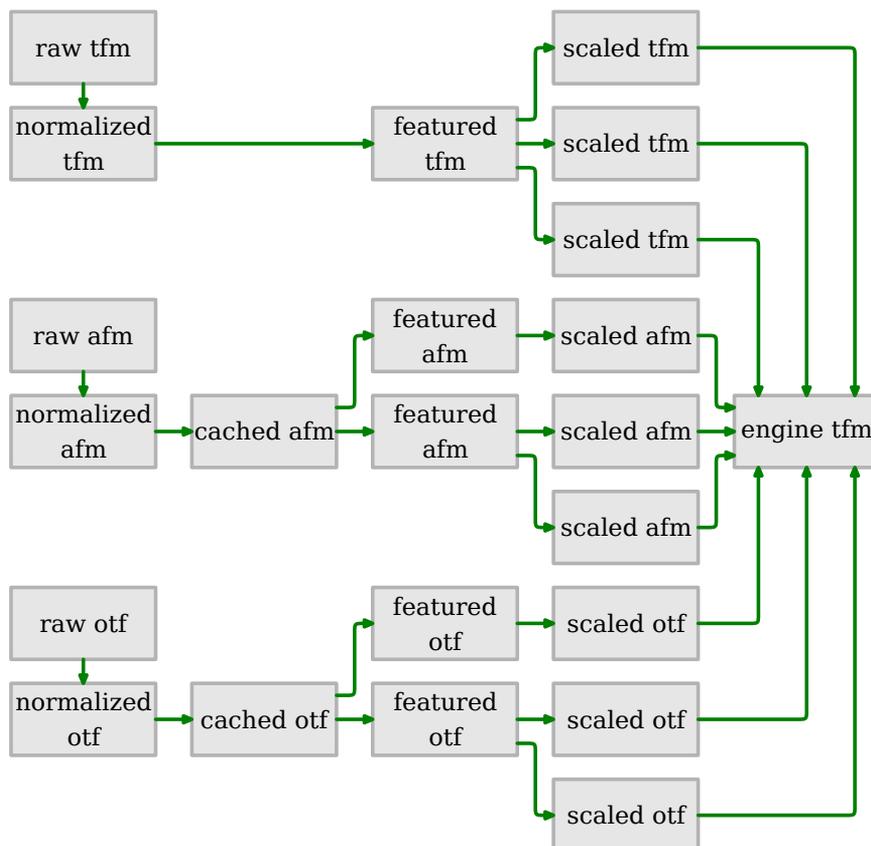
Figure 2.1 Defining a font.

The first step is loading the font (or using the cached copy). From that a copy is made that has some additional data concerning the features set and from that a scaled copy is constructed. These copies share as much data as possible to keep the memory footprint as small as possible. The table that is passed to LuaTeX gets cleaned up afterwards. In practice the tfm loader only kicks in for creating virtual math fonts. The afm reader is used for Type1 fonts and as there is no free upgrade path from Type1 to OpenType for commercial fonts, that one will get used for older fonts. Of course most loading is done by the otf reader(s).

?? The data in the final tfm table is organized in subtables. The biggest ones are the `characters` and `descriptions` tables that have information about each glyph. Later we will see more of that. There are a few additional tables of which we show two: `properties` and `parameters`. For the current font the first one has the following entries:

```
autoitalicamount   <unset>
cidinfo            <unset>
embedding          subset
```

Modes

```
encodingbytes      2
filename           c:/data/develop/tex-context/tex/texmf/fonts/truetype/public/dejavu/
finalized          true
fontname           DejaVuSerif
format             truetype
fullname           DejaVu Serif
hasitalics         <unset>
hasmath            <unset>
mathitalics        <unset>
mode               node
name               DejaVuSerif
noglyphnames       true
nostackmath        <unset>
psname             DejaVuSerif
textitalics        <unset>
virtualized        <unset>
```

The parameters table has variables that have been (re)assigned in the process. A period
in the key indicates that we are dealing with a subtable, for instance expansion.

```
ascender                448128
descender               141696
designsize              655360
expansion.auto          <unset>
expansion.shrink        0
expansion.step          0
expansion.stretch       0
extendfactor            1000
factor                  288
hfactor                 288
mathsize                0
protrusion.auto         <unset>
quad                    589824
scaledpoints            589824
scriptpercentage        <unset>
scriptscriptpercentage  <unset>
size                    589824
slantfactor             0
slantperpoint           0
spacing.extra           62496
spacing.shrink          62496
spacing.stretch         93744
spacing.width           187488
units                   2048
```

```
vfactor                 288
xheight                 306144
```

To give you an impression of what we are dealing with, the positional features are shown next:

The substitution features of the current font are as follows:

```
aalt              cyrl dflt абвгдежзийклмнопрстуфхцчшщъыьѣэюяѳ
                           АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬѢЭЮЯѲ
                           1234567890 1/2
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              cyrl mkd  абвгдежзийклмнопрстуфхцчшщъыьѣэюяѳ
                           АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬѢЭЮЯѲ
                           1234567890 1/2
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              cyrl srb  абвгдежзийклмнопрстуфхцчшщъыьѣэюяѳ
                           АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬѢЭЮЯѲ
                           1234567890 1/2
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              dflt dflt abcdefghijklmnopqrstuvwxyz
                           1234567890 1/2
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              grek dflt 1234567890 1/2
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              latn aze  abcdefghijklmnopqrstuvwxyz
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              latn crt  abcdefghijklmnopqrstuvwxyz
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              latn dflt abcdefghijklmnopqrstuvwxyz
                           1234567890 1/2
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              latn gag  abcdefghijklmnopqrstuvwxyz
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              latn ism  abcdefghijklmnopqrstuvwxyz
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              latn kaz  abcdefghijklmnopqrstuvwxyz
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              latn krk  abcdefghijklmnopqrstuvwxyz
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              latn ksm  abcdefghijklmnopqrstuvwxyz
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              latn lsm  abcdefghijklmnopqrstuvwxyz
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
aalt              latn mol  abcdefghijklmnopqrstuvwxyz
                           . , : ; ? ! ‹ › « »  @ # $ % & * () [] {} <> + - = / |
```

Modes

aalt	latn	nsm	abcdefghijklmnopqrstuvwxyz
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
aalt	latn	rom	abcdefghijklmnopqrstuvwxyz
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
aalt	latn	sks	abcdefghijklmnopqrstuvwxyz
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
aalt	latn	ssm	abcdefghijklmnopqrstuvwxyz
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
aalt	latn	tat	abcdefghijklmnopqrstuvwxyz
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
aalt	latn	trk	abcdefghijklmnopqrstuvwxyz
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
ccmp	cyrl	dflt	абвгдежзийклмнопрстуфхцчшщъыьэюяѳ
			АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯѲ
			1234567890 1/2
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
ccmp	cyrl	mkd	абвгдежзийклмнопрстуфхцчшщъыьэюяѳ
			АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯѲ
			1234567890 1/2
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
ccmp	cyrl	srb	абвгдежзийклмнопрстуфхцчшщъыьэюяѳ
			АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯѲ
			1234567890 1/2
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
ccmp	dflt	dflt	abcdefghijklmnopqrstuvwxyz
			1234567890 1/2
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
ccmp	grek	dflt	1234567890 1/2
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
ccmp	latn	aze	abcdefghijklmnopqrstuvwxyz
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
ccmp	latn	crt	abcdefghijklmnopqrstuvwxyz
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
ccmp	latn	dflt	abcdefghijklmnopqrstuvwxyz
			1234567890 1/2
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
ccmp	latn	gag	abcdefghijklmnopqrstuvwxyz
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
ccmp	latn	ism	abcdefghijklmnopqrstuvwxyz
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
ccmp	latn	kaz	abcdefghijklmnopqrstuvwxyz
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |
ccmp	latn	krk	abcdefghijklmnopqrstuvwxyz
			. , : ; ? ! ‹ › « » @ # $ % & * () [] {} <> + - = / |

ccmp	latn	ksm	abcdefghijklmnopqrstuvwxyz
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
ccmp	latn	lsm	abcdefghijklmnopqrstuvwxyz
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
ccmp	latn	mol	abcdefghijklmnopqrstuvwxyz
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
ccmp	latn	nsm	abcdefghijklmnopqrstuvwxyz
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
ccmp	latn	rom	abcdefghijklmnopqrstuvwxyz
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
ccmp	latn	sks	abcdefghijklmnopqrstuvwxyz
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
ccmp	latn	ssm	abcdefghijklmnopqrstuvwxyz
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
ccmp	latn	tat	abcdefghijklmnopqrstuvwxyz
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
ccmp	latn	trk	abcdefghijklmnopqrstuvwxyz
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
char-ligatures	*	*	abcdefghijklmnopqrstuvwxyz
			1234567890 1/2
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
compat-ligatures	*	*	abcdefghijklmnopqrstuvwxyz
			1234567890 1/2
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
dlig	cyrl	dflt	абвгдежзийклмнопрстуфхцчшщъыљэюяѳ
			АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬѢЭЮЯѲ
			1234567890 1/2
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
dlig	cyrl	mkd	абвгдежзийклмнопрстуфхцчшщъыљэюяѳ
			АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬѢЭЮЯѲ
			1234567890 1/2
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
dlig	cyrl	srb	абвгдежзийклмнопрстуфхцчшщъыљэюяѳ
			АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬѢЭЮЯѲ
			1234567890 1/2
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
dlig	dflt	dflt	abcdefghijklmnopqrstuvwxyz
			1234567890 1/2
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
dlig	grek	dflt	1234567890 1/2
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |
dlig	latn	aze	abcdefghijklmnopqrstuvwxyz
			., : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

dlig		latn	crt	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	dflt	abcdefghijklmnopqrstuvwxyz 1234567890 1/2 . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	gag	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	ism	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	kaz	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	krk	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	ksm	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	lsm	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	mol	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	nsm	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	rom	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	sks	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	ssm	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	tat	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
dlig		latn	trk	abcdefghijklmnopqrstuvwxyz . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
liga		cyrl	dflt	абвгдежзийклмнопрстуфхцчшщъыьэюяѳ АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯѲ 1234567890 1/2 . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
liga		cyrl	mkd	абвгдежзийклмнопрстуфхцчшщъыьэюяѳ АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯѲ 1234567890 1/2 . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
liga		cyrl	srb	абвгдежзийклмнопрстуфхцчшщъыьэюяѳ АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯѲ 1234567890 1/2 . , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |

```
liga            dflt  dflt  abcdefghijklmnopqrstuvwxyz
                            1234567890 1/2
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            grek  dflt  1234567890 1/2
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  aze   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  crt   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  dflt  abcdefghijklmnopqrstuvwxyz
                            1234567890 1/2
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  gag   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  ism   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  kaz   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  krk   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  ksm   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  lsm   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  mol   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  nsm   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  rom   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  sks   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  ssm   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  tat   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
liga            latn  trk   abcdefghijklmnopqrstuvwxyz
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
locl            cyrl  mkd   абвгдежзийклмнопрстуфхцчшщъыьѐэюяѳ
                            АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЀЭЮЯѲ
                            1234567890 1/2
                            . , : ; ? ! ‹ › « »   @ # $ % & * ( ) [ ] { } <> + - = / |
locl            cyrl  srb   абвгдежзийклмнопрстуфхцчшщъыьѐэюяѳ
                            АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЀЭЮЯѲ
```


1234567890 1/2
. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

locl		latn	ism	abcdefghijklmnopqrstuvwxyz

. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| locl | | latn | ksm | abcdefghijklmnopqrstuvwxyz |

. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| locl | | latn | lsm | abcdefghijklmnopqrstuvwxyz |

. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| locl | | latn | nsm | abcdefghijklmnopqrstuvwxyz |

. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| locl | | latn | sks | abcdefghijklmnopqrstuvwxyz |

. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| locl | | latn | ssm | abcdefghijklmnopqrstuvwxyz |

. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| salt | | cyrl | dflt | абвгдежзийклмнопрстуфхцчшщъыьѣэюяѳ |

АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬѢЭЮЯѲ
1234567890 1/2
. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| salt | | cyrl | mkd | абвгдежзийклмнопрстуфхцчшщъыьѣэюяѳ |

АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬѢЭЮЯѲ
1234567890 1/2
. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| salt | | cyrl | srb | абвгдежзийклмнопрстуфхцчшщъыьѣэюяѳ |

АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬѢЭЮЯѲ
1234567890 1/2
. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| salt | | dflt | dflt | abcdefghijklmnopqrstuvwxyz |

1234567890 1/2
. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| salt | | grek | dflt | 1234567890 1/2 |

. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| salt | | latn | aze | abcdefghijklmnopqrstuvwxyz |

. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| salt | | latn | crt | abcdefghijklmnopqrstuvwxyz |

. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| salt | | latn | dflt | abcdefghijklmnopqrstuvwxyz |

1234567890 1/2
. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| salt | | latn | gag | abcdefghijklmnopqrstuvwxyz |

. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| salt | | latn | ism | abcdefghijklmnopqrstuvwxyz |

. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

| salt | | latn | kaz | abcdefghijklmnopqrstuvwxyz |

. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } <> + - = / |

salt		latn	krk	abcdefghijklmnopqrstuvwxyz
				. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
salt		latn	ksm	abcdefghijklmnopqrstuvwxyz
				. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
salt		latn	lsm	abcdefghijklmnopqrstuvwxyz
				. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
salt		latn	mol	abcdefghijklmnopqrstuvwxyz
				. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
salt		latn	nsm	abcdefghijklmnopqrstuvwxyz
				. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
salt		latn	rom	abcdefghijklmnopqrstuvwxyz
				. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
salt		latn	sks	abcdefghijklmnopqrstuvwxyz
				. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
salt		latn	ssm	abcdefghijklmnopqrstuvwxyz
				. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
salt		latn	tat	abcdefghijklmnopqrstuvwxyz
				. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
salt		latn	trk	abcdefghijklmnopqrstuvwxyz
				. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
tlig		*	*	abcdefghijklmnopqrstuvwxyz
				1234567890 1/2
				. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |
trep		*	*	abcdefghijklmnopqrstuvwxyz
				1234567890 1/2
				. , : ; ? ! ‹ › « » @ # $ % & * ( ) [ ] { } < > + - = / |

This is clearly an OpenType font. Normally there are a default script and default language supported. If this is not the case you need to provide them as part of the feature-set, otherwise there will be no features applied.

2.3 Base mode

We talk of base mode processing when the font machinery is used that is built in LuaTeX. So what does this traditional mechanism provide?

Before we discuss this, a somewhat simplified model of how TeX works has to be given. Say that we have the following input:

```
\def\bla{BLA}
test \bla test
```

This input gets translated into tokens and those tokens are either processed later or they become something else directly. Take the first line. Characters in the input have a so called catcode property that determines how the parser tokenized them. Effectively we therefore get something like this:

```
<command def>
<command bla>
<begingroup>
<character B>
<character L>
<character A>
<endgroup>
```

and finally in the hash table there will be an entry for `bla` that has the meaning BLA expressed in three characters.

The second line refers to `\bla` and in the process this macro gets expanded, so we get:

```
<character t>
<character e>
<character s>
<character t>
<space>
<character B>
<character L>
<character A>
<character t>
<character e>
<character s>
<character t>
```

Because the parser gobbles spaces after a macro name, there is no space before the second `test`. In practice there will be no intermediate list like this, because as soon as possible TEX will add something to a so called node list. When the moment is there, this list will be passed to the typesetting routine that constructs a horizontal list. Later this list can be converted into a horizontal box or broken into lines when it concerns a paragraph.

In traditional TEX characters are stored into char nodes and the builder turns them into glyph nodes. In LuaTEX they start out as glyph nodes and the subtype number will flag them as glyphs. Any value larger than 255 is a signal that the list has been processed. The previous example leads to the list shown in figure 2.2.

Here we have turned off inter-character kerning and hyphenation. When we turn that on, we get a slightly more complex list, as shown in figure 2.3. Hyphenation points are represented by discretionary nodes and these have pointers to a pre break, post break and replacement text.

In addition to hyphenation and kerning we can have ligatures. The list in figure 2.4 shows that we get a reference to a ligature in the glyph node but that the components are still known. This figure also demonstrates that the ligature is build in steps.
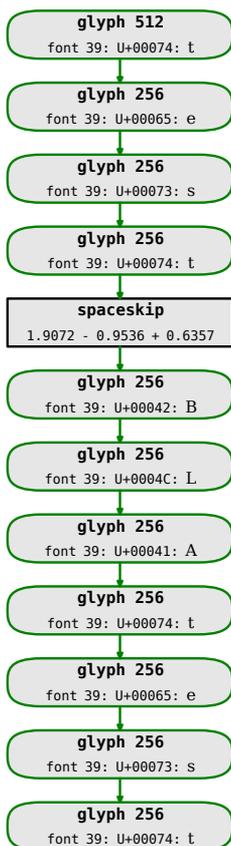
Figure 2.2   The text 'test BLAtest'
converted to nodes.

If we insert an explicit \discretionary command, we see in figure 2.5 that we get three variants. In figure 2.6 we render some Dutch words and these have quite some ligatures.

So, we have hyphenation, ligature building and kerning and to some extent these mechanisms hook into each other. This process is driven by information stored in the font and rules related to the language. The hyphenation happens first, so the builder just sees discretionary nodes and needs to act properly on them. Although languages play an important role in formatting the text, for the moment we can forget about that. This leaves the font.

As we already mentioned in a previous chapter, in ConTEXt we use Unicode internally. This also means that fonts are organized this way. By default the glyph representation of a Unicode character sits in the same slot in the glyph table. All additional glyphs, like ligatures or alternates are pushed in the private unicode space. This is why in the lists shown in the figures the ligatures have a private Unicode number.

The basic mode of operation in the builder in LuaTEX is as follows:
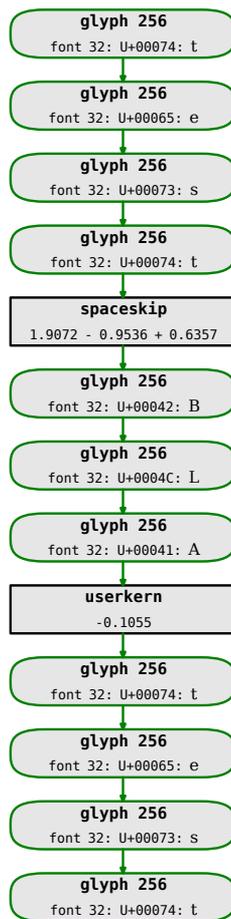
Figure 2.3 The text 'test BLAtest' converted to nodes, hyphenated and kerned.

- hyphenate the node list
- build ligatures
- inject kerns
- optionally break into lines

In traditional TeX the first step is not that independent. There hyphenation takes place when the text is broken into lines, and only in places that are candidate for such a break. In LuaTeX the whole text is hyphenated. This has the advantage that the steps are clearly separated and that no complex reconstruction and re-hyphenation has to take place. The speed penalty can be neglected and the extra memory overhead is small compared to what is needed anyway.

In base mode the raw font data is read in and from that only basic information is used to

```
glyph 256
font 32: U+00061: a

disc                    prebreak : glyph 1      postbreak : glyph 1      replacement : glyph 2
                        font 32: U+00066: f      font 32: U+00066: f       font 32: U+0FB00: ﬀ

glyph 256               userkern
font 32: U+00069: i     -0.2139

glyph 256               prebreak : glyph 1
font 32: U+0006C: l     font 32: U+0002D: -

disc                    prebreak : glyph 1
                        font 32: U+0002D: -

glyph 256
font 32: U+00069: i

disc                    prebreak : glyph 1
                        font 32: U+0002D: -

glyph 256
font 32: U+00061: a

disc                    prebreak : glyph 1
                        font 32: U+0002D: -

glyph 256
font 32: U+00074: t

glyph 256
font 32: U+00069: i

glyph 256
font 32: U+0006F: o

glyph 256
font 32: U+0006E: n
```

Figure 2.4 The rendering of the word ‘`affiliation`’.

construct the tfm table: dimensions, ligatures and kerns. In a node list, all glyph ranges that refer to such a font get the standard ligature and kern routines applied, but only if the subtype is still less than 256. This check on subtype prevents duplicate processing that might happen as a side effect of for instance unboxing some material in a yet to be typeset text.

Given that the majority of what TEX has to deal with is relatively simple latin script, base mode processing is rather convenient and efficient. It is also the reference point of other kinds of processing. The most simple way to force base mode is the following:

```
\definefontfeature[basemode][mode=base,kern=yes,liga=yes]
```

```
\definefont[MyTitleFont][SerifBold*basemode at 12pt]
```

Here \MyTitleFont will be a bold serif with ligatures and kerns applied. However, as an OpenType font can have many features, the following definitions are also valid:
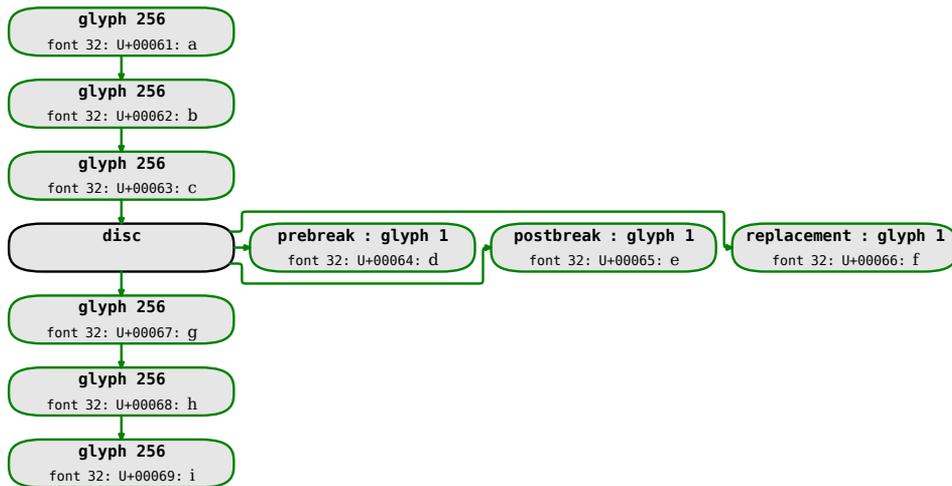
Figure 2.5 The rendering of the bogus word 'abcghi' with an explicit discretionary added.

```
\definefontfeature[basemode-o][mode=base,kern=yes,onum=yes,liga=yes]
\definefontfeature[basemode-s][mode=base,kern=yes,smcp=yes]
```

The tfm constructor will filter the right information from the font data and construct a proper table based on these specifications. But you need to keep in mind that when for instance old style numerals or small caps are activated, that their rendering (the glyph) will always be used. So, for instance 3 and A keep their Unicode points but as part of their specification they will get an index pointing to the oldstyle or small caps variant and the dimensions of that shape will be used.

2.4 Node mode

Node mode is by far the most interesting of the modes. When enabled we only pass a few properties of glyphs to the engine: the width, height and depth and optionally protrusion, expansion factors as well as some extra ConTEXt specific quantities. So there is no kerning and no ligature building done by the engine. Instead we do this in Lua by walking over the node list and checking if some action is needed.

The default feature set enables kerning and ligature building for default and/or Latin ?? scripts and the default language. Being a relative simple feature, ligatures don't take much action. Next we show a trace of a ligature replacement.

font   43: DejaVuSerif.ttf @ 24.0pt
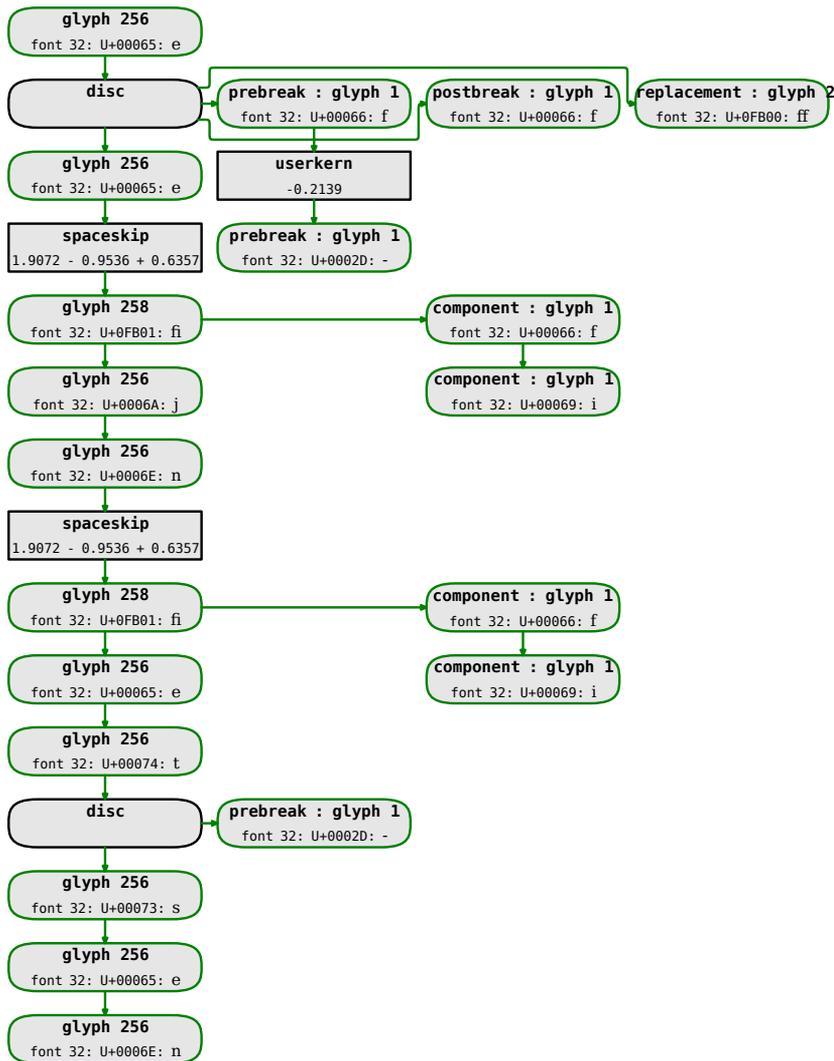
Figure 2.6 The rendering of the Dutch words ' effe fijn fietsen'.

features analyze=yes, autolanguage=position, autoscript=position,
checkmarks=yes, curs=yes, devanagari=yes, dummies=yes,
extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes,
liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes,
mode=node, script=dflt, spacekern=yes, tlig=yes, trep=yes

step 1 affiliation [+TLT] U+61:a U+66:f U+66:f U+69:i
U+6C:l U+69:i U+61:a U+74:t U+69:i U+6F:o U+6E:n

feature 'liga', type 'gsub_ligature', lookup 's_s_3', replacing

```
U+00066 (f) upto U+00066 (f) by ligature U+0FB00 case 2
```

result affiliation [+TLT] U+61:a U+FB00:ff U+69:i U+6C:l U+69:i U+61:a U+74:t U+69:i U+6F:o U+6E:n

Be warned that this f f i sequence not always becomes a ligature. Actually this is one area where tradition is quite visible: for some reason most fonts do have these f-related ligatures but lack others. These ligatures even have code points in Unicode which is quite debatable. Just as there are fonts with hardly any kerns (like Lucida) there are fonts that follow a different route to improve the look and feel of neighbouring glyphs, like Cambria:

font   45: cambria.ttf @ 24.0pt

features   analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes, mode=node, script=latn, spacekern=yes, tlig=yes, trep=yes

step 1 affiliation [+TLT] U+61:a U+66:f U+66:f U+69:i U+6C:l U+69:i U+61:a U+74:t U+69:i U+6F:o U+6E:n

feature 'liga', type 'gsub_contextchain', chain lookup 's_s_17', index 1, replacing single U+00066 by U+F016D

result affiliation [+TLT] U+61:a U+66:f U+F016D:f U+69:i U+6C:l U+69:i U+61:a U+74:t U+69:i U+6F:o U+6E:n

Instead of representing multiple characters by one glyph the designer has decided to replace the f by a slightly narrower one so that the dot of the i stays loose.

An example where much more is involved is the following. The Husayni font that is used for typesetting Arabic is built upon a solid but complex OpenType foundation and can only be dealt with in node mode. When the LuaTEX project started we assumed that more power in the engine was needed to accomplish this, but so far the results with standard OpenType functionality are quite good. ConTEXt has an additional paragraph optimizer that can apply additional features to get even better results but discussing this falls beyond this chapter. A trace of just one Arabic word is much longer than the previously shown traces.

font   46: husayni.ttf @ 48.0pt

features   analyze=yes, anum=yes, calt=yes, ccmp=yes, checkmarks=yes,
curs=yes, devanagari=yes, dummies=yes, extensions=yes,
extrafeatures=yes, extraprivates=yes, fina=yes, init=yes,
js16=yes, kern=yes, language=dflt, mark=yes, mathkerns=yes,
mathrules=yes, medi=yes, mkmk=yes, mode=node, rlig=yes,
salt=yes, script=arab, spacekern=yes, ss01=yes, ss03=yes,
ss10=yes, ss12=yes, ss15=yes, ss16=yes, ss19=yes, ss24=yes,
ss25=yes, ss26=yes, ss27=yes, ss31=yes, ss34=yes, ss35=yes,
ss36=yes, ss37=yes, ss38=yes, ss41=yes, ss42=yes, ss43=yes,
ss60=yes, tlig=yes
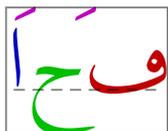
step 1 [+TRT] U+641: U+64E: U+62E: U+64E:

U+627:

feature 'ccmp', type 'gsub_multiple', lookup 's_s_15',
replacing U+00641 (Faa) by multiple U+00641 U+0FBB2 (Faa
Onedotabove)

feature 'ccmp', type 'gsub_multiple', lookup 's_s_15',
replacing U+0062E (Khaa) by multiple U+0062E U+0FBB2 (Khaa
Onedotabove)

step 2 [+TRT] U+641: U+FBB2: U+64E: U+62E:
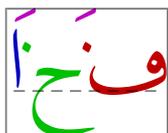
U+FBB2: U+64E: U+627:

feature 'fina', type 'gsub_alternate', lookup 's_s_20',
replacing U+00627 (Alif) by alternative 'U+F0006
(Alif.final)' to value 1, taking 1,

step 3 [+TRT] U+641: U+FBB2: U+64E: U+62E:
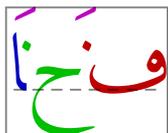
U+FBB2: U+64E: U+F0006:

feature 'medi', type 'gsub_single', lookup 's_s_21', replacing
U+0062E (Khaa) by single U+F001E (Khaa.medial)

Modes
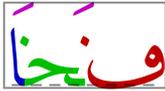
step 4 [+TRT] U+641: U+FBB2: U+64E: U+F001E:

U+FBB2: U+64E: U+F0006:

feature 'init', type 'gsub_single', lookup 's_s_22', replacing
 U+00641 (Faa) by single U+F003D (Faa.initial)

step 5 [+TRT] U+F003D: U+FBB2: U+64E: U+F001E: U+FBB2:

U+64E: U+F0006:

feature 'ss03', type 'gsub_contextchain', chain lookup
 's_s_59', index -1, replacing single U+F003D (Faa.initial) by
 U+F029D (Faa.FJ_im)

feature 'ss03', type 'gsub_contextchain', chain lookup
 's_s_59', index -1, replacing single U+F001E (Khaa.medial) by
 U+F02D8 (Khaa.LJ_im)

step 6 [+TRT] U+F029D: U+FBB2: U+64E: U+F02D8: U+FBB2: U+64E:

U+F0006:

feature 'rlig', type 'gsub_contextchain', chain lookup
 's_s_174', replacing U+F02D8 (Khaa.LJ_im) by multiple
 characters U+F02D8 U+00640 (Khaa.LJ_im Tatwiil)

step 7 [+TRT] U+F029D: U+FBB2: U+64E: U+F02D8: U+640: U+FBB2:

U+64E: U+F0006:

feature 'salt', type 'gsub_contextchain', chain lookup
 's_s_186', index -1, replacing character U+F02D8 (Khaa.LJ_im)
 upto U+00640 (Tatwiil) by ligature U+F051B (Khaa.LJ_im_j1)
 case 4

step 8 [+TRT] U+F029D: U+FBB2: U+64E: U+F051B: U+FBB2: U+64E:

U+F0006:

feature 'salt', type 'gsub_contextchain', chain lookup
 's_s_225', index -1, replacing single U+F0006 (Alif.final) by
 U+F0302 (Alif.alt2_final)

step 9 [+TRT] U+F029D: U+FBB2: U+64E: U+F051B: U+FBB2: U+64E:

U+F0302:

feature 'curs', type 'gpos_cursive', lookup 'p_s_0', moving
 U+F029D (Faa.FJ_im) to U+F051B (Khaa.LJ_im_j1) cursive
 (0pt,9.84375pt) using anchor and bound 1 in r2l mode

feature 'curs', type 'gpos_cursive', lookup 'p_s_0', moving
 U+F051B (Khaa.LJ_im_j1) to U+F0302 (Alif.alt2_final) cursive
 (0pt,0pt) using anchor and bound 2 in r2l mode

step 10 [+TRT] U+F029D: U+FBB2: U+64E: U+F051B: U+FBB2: U+64E:

U+F0302:

feature 'ccmp', type 'gpos_mark2base', lookup 'p_s_16', anchor
 , bound 1, anchoring mark U+0FBB2 (Onedotabove) to basechar
 U+F029D (Faa.FJ_im) => (1.89844pt,1.59375pt)

feature 'ccmp', type 'gpos_mark2base', lookup 'p_s_16', anchor
 , bound 2, anchoring mark U+0FBB2 (Onedotabove) to basechar
 U+F051B (Khaa.LJ_im_j1) => (5.48438pt,0.11719pt)

step 11 [+TRT] U+F029D: U+FBB2: U+64E: U+F051B: U+FBB2: U+64E:

U+F0302:

feature 'mark', type 'gpos_mark2base', lookup 'p_s_16', anchor
 , bound 3, anchoring mark U+0FBB2 (Onedotabove) to basechar
 U+F029D (Faa.FJ_im) => (1.89844pt,1.59375pt)

feature 'mark', type 'gpos_mark2base', lookup 'p_s_16', anchor
 , bound 4, anchoring mark U+0FBB2 (Onedotabove) to basechar
 U+F051B (Khaa.LJ_im_j1) => (5.48438pt,0.11719pt)

step 12 [+TRT] U+F029D: U+FBB2: U+64E: U+F051B: U+FBB2: U+64E:

U+F0302:

Modes

```
feature 'mark', type 'gpos_mark2base', lookup 'p_s_26', anchor
  , bound 5, anchoring mark U+0064E (Fathah) to basechar
  U+F029D (Faa.FJ_im) => (5.46094pt,-2.92969pt)

feature 'mark', type 'gpos_mark2base', lookup 'p_s_26', anchor
  , bound 6, anchoring mark U+0064E (Fathah) to basechar
  U+F051B (Khaa.LJ_im_j1) => (6.39844pt,-4.35938pt)
```

result [+TRT] U+F029D: U+FBB2: U+64E: U+F051B: U+FBB2: U+64E: U+F0302:
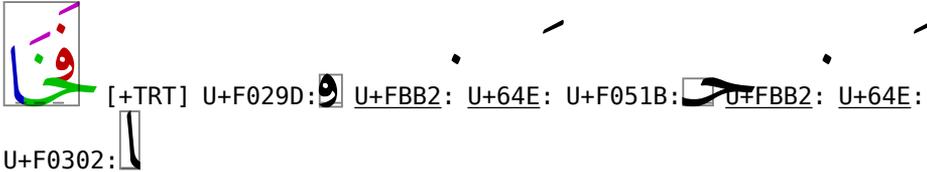
What we see here is a stepwise substitution process, sometimes based on a contextual analysis, followed by positioning. The coloring concerns the outcome of the analysis which in this case flags initial, final, medial and isolated characters.

The starting point of this Arabic word is visualized in figure 2.7 and as expected we see no discretionary nodes here. The result as seen in figure 2.8 has (interestingly) no kerns as all replacements happen via offsets in the glyph node.
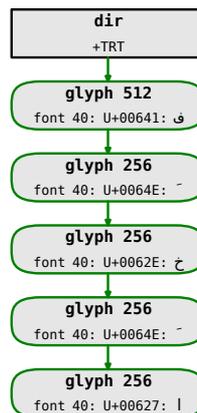
Figure 2.7   The Arabic input 'خَﺍَﻓ'
before rendering.

2.5 Auto mode

Base mode is lean and mean and relatively fast while node mode is more powerful and slower. So how do you know what to choose? The safest bet is to use node mode for everything. In ConTeXt however, we also have the so called auto mode. In that case there is some analysis going on that chooses between base mode and node mode depending on the boundary conditions of script and language and there are specific demands in terms of feature processing. So, auto mode will resolve to base or node mode.
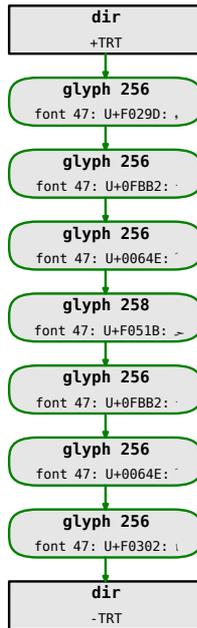
Figure 2.8   The Arabic input 'ﻑ َ ﺥ َ ﺍ' after rendering.

2.6 None mode

Sometimes no features have to be applied at all. A good example is verbatim. There you don't want ligatures, kerning or fancy substitutions. Contrary to what you might expect, monospaced fonts can have such features. Some might actually make sense, like rendering zeros. However, you cannot assume such a feature to be present so this is an example of where some more knowledge about a particular font is needed. This is what Latin Modern provides.

Normally using mode none for situations that need to be predictable is quite okay.

2.7 **Dynamics**

Sometimes you want to enable or disable a specific feature only for a specific span of text. Defining a font for only this occasion is overkill, especially when for instance features are used to fine-tune the typography as happens in the Oriental TEX project, which is related to LuaTEX. Instead of defining yet another font instance we can therefore enable and disable specific features. For this it is not needed to know the current font and its size.[4]

Dynamics are a special case of node mode and you don't need to set it up when defining a font. In fact, a font defined in base mode can also be dynamic. We show some simple examples of applying dynamic features.

Let's first define some feature sets:

```
\definefontfeature[f:smallcaps][smcp=yes]
\definefontfeature[f:nocaps]   [smcp=no]
\definefontfeature[f:oldstyle] [onum=yes]
\definefontfeature[f:newstyle] [onum=no]
```

We can add and subtract these features from the current feature set that is bound to the current font.

```
\switchtobodyfont[pagella]    123 normal
\addfeature      {f:oldstyle}  123 oldstyle
\addfeature      {f:smallcaps} 123 olstyle smallcaps
\subtractfeature{f:oldstyle}  123 smallcaps
\subtractfeature{f:smallcaps} 123 normal
```

Here we choose a font that has oldstyle numerals as well as small caps: pagella.

123 normal 123 oldstyle 123 ᴏʟꜱᴛʏʟᴇ ꜱᴍᴀʟʟᴄᴀᴘꜱ 123 ꜱᴍᴀʟʟᴄᴀᴘꜱ 123 normal

The following does the same, but only uses addition:

```
\switchtobodyfont[pagella] 123 normal
\addfeature{f:oldstyle}    123 oldstyle
\addfeature{f:smallcaps}   123 olstyle smallcaps
\addfeature{f:newstyle}    123 smallcaps
\addfeature{f:nocaps}      123 normal
```

You can also completely replace a feature set. Of course the set is only forgotten inside the current group.

```
\switchtobodyfont[pagella]    123 normal
```

---

[4] Dynamics are a ConTEXt specific feature and is not available in the generic version of the font code. There are several reasons for this: it complicates the code, it assumes the ConTEXt feature definition mechanism to be used, and it is somewhat slower as some extra analysis has to be done.

```
\addfeature     {f:oldstyle}  123 oldstyle
\addfeature     {f:smallcaps} 123 olstyle smallcaps
\replacefeature{f:oldstyle}   123 oldstyle
\replacefeature{f:smallcaps} 123 smallcaps
```

and now we get:

123 normal 123 oldstyle 123 ᴏʟsᴛʏʟᴇ sᴍᴀʟʟᴄᴀᴘs 123 oldstyle 123 sᴍᴀʟʟᴄᴀᴘs

You can exercise some control with \resetfeature:

```
\switchtobodyfont[pagella]    123 normal
\addfeature     [f:oldstyle]  123 oldstyle
\addfeature     [f:smallcaps] 123 olstyle smallcaps
\resetfeature                 123 reset
\addfeature     [f:oldstyle]  123 oldstyle
\addfeature     [f:smallcaps] 123 olstyle smallcaps
```

Watch how we use the [] variant of the commands. The braced and bracketed variants behave the same.

123 normal 123 oldstyle 123 ᴏʟsᴛʏʟᴇ sᴍᴀʟʟᴄᴀᴘs 123 reset 123 oldstyle 123 ᴏʟsᴛʏʟᴇ sᴍᴀʟʟᴄᴀᴘs

There is also a generic command \feature that takes two arguments. Below we show all calls, with long and short variants:

```
\addfeature         [f:mine] \feature [more][f:mine] \feature[+][f:mine]
\subtractfeature    [f:mine] \feature [less][f:mine] \feature[-][f:mine]
\replacefeature     [f:mine] \feature  [new][f:mine] \feature[=][f:mine]
\resetandaddfeature[f:mine] \feature[local][f:mine] \feature[!][f:mine]
\revivefeature      [f:mine] \feature  [old][f:mine] \feature[>][f:mine]
\resetfeature                \feature[reset]         \feature[<]
```

Each variant also accepts {} instead of [] so that they can conveniently be used in square bracket arguments. As a bonus, the following also works:

```
\switchtobodyfont[pagella]
123 normal
\feature[+][f:smallcaps,f:oldstyle]
123 SmallCaps and OldStyle
```

Here is the proof:

123 normal 123 SᴍᴀʟʟCᴀᴘs ᴀɴᴅ OʟᴅSᴛʏʟᴇ

2.8 Discretionaries

One of the complications in supporting more complex features is that we can have discretionary nodes. These are either inserted by the hyphenation engine, or explicitly by the

user (directly or via macros). In most cases we don't need to bother about this. For instance, more demanding scripts like Arabic don't hyphenate, languages using the Latin script seldom want ligatures at hyphenation points (as they can be compound words) and/or avoid confusing hyphenation points, so what is left are specific user inserted discretionaries. Add to that, that a proper font has not much kerning between lowercase characters and it will be clear that we can ignore most of this. Anyway, as we explicitly deal with user discretionaries, the next works out okay. Watch how we normally only have something special in the replacements text that shows up when no hyphenation is needed.

```
\language[nl]
\definedfont[file:texgyrepagella-regular.otf*default]
\hsize  1mm vereffenen \par
\hsize  1mm effe \par
\hsize  1mm e\discretionary{f-}{f}{ff}e \par
\hsize 20mm e\discretionary{f-}{f}{ff}e \par
\smallcaps
\hsize  1mm vereffenen \par
\hsize  1mm effe \par
\hsize  1mm e\discretionary{f-}{f}{ff}e \par
\hsize 20mm e\discretionary{f-}{f}{ff}e \par
```

ver-	ef-	fe	VER-	NEN	EF-
ef-	fe		EF-		FE
fe-		effe	FE-	EF-	
nen	ef-			FE	EFFE

In base mode such things are handled by the TeX engine itself and it can deal with pretty complex cases. In node mode we use a simplification which in practice suffices. We will come back to this in section 5.2.12.

2.9 Efficiency

The efficiency of the mechanisms described here depends on several factors. It will be clear that the larger the font, the more time it will take to load it. But what is large? Most cjk fonts are pretty large but also rather simple. A font like Zapfino on the other hand covers only latin but comes with many alternative shapes and a large set of rules. The Husayni font focusses on Arabic, which in itself has not that large an alphabet, but being an advanced script font, it has a lot of features and definitely a lot of rules.

In terms of processing it's safe to say that Latin is of average complexity. At most you will get some substitutions, like regular numerals being replaced by oldstyles, or ligature building, which involves a bit of analysis, and some kerning at the end. In base mode the substitutions have no overhead, simply because the character table already has references to the substituents and the replacement already takes place when defining the

font. There ligature building and kerning are also fast because of the limited amount of lookups that also are already kept with the characters. In node mode however, the lists have to be parsed and tables have to be consulted so even Latin processing has some overhead: each glyph node is consulted and analyzed (either or not in its context), often multiple times. However, the code is rather optimized and we use caching of already analyzed data when possible.

A cjk script is somewhat more complex on the one hand, but pretty simple on the other. Instead of font based kerning, we need to prevent or encourage breaks between certain characters. This information is not in the font and is processed otherwise but it does cost some time. The font part however is largely idle as there are no features to be applied. Even better, because the glyphs are large and the information density is high, the processing time per page is not much different from Latin. Base mode is good enough for most cjk.

The Arabic script is another matter. There we definitely go beyond what base mode offers so we always end up in node mode. Also, because there is some analysis involved, quite some substitutions and in the end also positioning, these are the least efficient fonts in terms of processing time. Of course the fact that we mix directions also plays a role. If in the Husayni font you enable 30 features with an average of 5 rules per feature, a 300 character paragraph will take 45.000 actions.[5] When multiple fonts are combined in a paragraph there will be more sweeps over the list and of course the replacements also have to happen.

In a time when the average photo camera produces megabyte pictures it makes no sense to whine about the size of a font file. On the other hand as each font eventually ends up in memory as a Lua table, it makes sense to optimize that bit. This is why fonts are converted into a more efficient intermediate table that is cached on disk. This makes loading a font quite fast and due to shared tables memory usage rather efficient. Of course a scaled instance has to be generated too, but that is acceptable. To some extent loading and defining a font also depends on the way the macro package is set up.

When comparing LuaTEX with for instance pdfTEX or XƎTEX you need to take into account that in ConTEXt MkIV we tend to use OpenType fonts only so there are less fonts loaded than in a more traditional setup. In ConTEXt startup time of MkIV is less than MkII although overall processing time is slower, which is due to Unicode being used and more functionality being provided. On the other hand, immediate MetaPost processing and more clever multipass handling wins back time. The impact of fonts on processing time in a regular document is therefore not that impressive. In practice a MkIV run can be faster than a MkII run, especially when MetaPost is used.

In ConTEXt processing of node lists with respect to fonts is only one of the many manipulations of such lists and by now fonts are not really the bottleneck. The more not font

---

[5] For a modern machine this amount is no real issue, but as each action involves function calls and possibly some garbage collection there is some price to pay.

related features users demand and enable, the less the relative impact of font processing becomes.

Also, there are some advanced typographic extras that LuaTeX offers, like protrusion (think of hanging punctuation) and hz optimization (glyph scaling) and these slow down processing quite a lot, and they are not taking place at the Lua end at all, but this might change in MkIV. And, of course, typesetting involves more than fonts and other aspects can be way more demanding.

3 Lookups

3.1 Introduction

In traditional TeX a font is defined by referring to its filename. A definition looks like this:

```
\font \MyFontA = lmr10
\font \MyFontB = lmr10 at 20pt
\font \MyFontC = lmr10 scaled 1500
```

The first definition defines the command `MyFontA` as a reference to the font stored in the file `lmx10`. No scaling takes place so the natural size is taken. This so called designsize is in no way standardized. Just look at these three specimen:

Design Size (Dejavu)

Design Size (Cambria)

Design Size (Latin Modern)

The designsize is normally 10 point, but as there is no real reference for this a designer decides how to translate this into a visual representation. As a consequence the `20pt` in the second line of the example definitions only means that the font is scaled to (normally) twice the designsize. The third line scaled by a factor `1.5` and the reason for using a value thousand times larger is that TeX's numbers are integers.

The next three lines are typical for Latin Modern (derived from Computer Modern) because this family comes in different design sizes.

```
\font \MyFontD = lmr12
\font \MyFontE = lmr12 at 20pt
\font \MyFontF = lmr12 scaled 1500
```

Because the designsize is part of the font metrics the second line (`\MyFontE`) is of similar size as `\MyFontB` although the 12 point variant is visually better suited for scaling up.

These definitions refer to files, but what file? What gets loaded is the file with name `name.tfm`. Eventually for embedding in the (let's assume pdf) file the outlines are taken from `name.pfb`. At that stage, when present, a `name.vf` is consulted in order to resolve characters that are combinations of others (potentially from other `pfb` files). The mapping from `name.tfm` to `name.pfb` filename happens in the so called map file. This means that one can also refer to another file, for instance `name.ttf`.

All this logic is hard coded in the engine and because the virtual font mechanism was introduced later without extending the `tfm` format, it can be hard at times to figure out issues when a (maybe obsolete) virtual file is present (this can be the case if you have generated the `tfm` file from an `afm` file that comes with the `pfb` file when you buy one.

But, in LuaTEX we no longer use traditional fonts and as a consequence we have more options open. Before we move on to them, we mention yet another definition:

```
\font \MyFontG = lmr12 sa 1.2
```

This method is not part of TEX but is provided by ConTEXt, MkII as well as MkIV. It means as much as "scale this font to 1.2 times the bodyfontsize". As this involves parsing the specification, it does not work as advertised here, but the next definition works okay:

```
\definefont[MyFontG][lmr12 sa 1.2]
```

This indicates that we already had a parser for font specifications on board which in turn made it relatively easy to do even more parsing, for instance for font features as introduced in XƎTEX and LuaTEX.

3.2 Specifications

In LuaTEX we intercept the font loader. We do so for several reasons.

- We want to make decisions on what file to load, this is needed when for instance there are files with the same name but different properties.
- We want to be able to lookup by file, by name, and by more abstract specification. In doing so, we want to be as tolerant as possible.
- We want to support several scaling methods, as discussed in the previous section.
- We want to implement several strategies for passing features and defining non standard approaches.

The formal specification of a font is as follows:

```
\definefont[PublicReference][filename]
\definefont[PublicReference][filename at dimension]
\definefont[PublicReference][filename scaled number]
```

We already had that extended to:

```
\definefont[PublicReference][filename]
\definefont[PublicReference][filename at dimension]
\definefont[PublicReference][filename scaled number]
\definefont[PublicReference][filename sa number]
```

So let's generalize that to:

```
\definefont[PublicReference][filename scaling]
```

And in MkIV we now have:

```
\definefont[PublicReference][filename*featurenames scaling]
\definefont[PublicReference][filename:featurespecication scaling]
```

```
\definefont[PublicReference][filename@virtualconstructor scaling]
```

The second variant is seldom used and is only provided because some users have fonts defined in the X∃TEX way. Users are advised not to use this method. The last method is special in the sense that it's used to define fonts that are constructed using the built in virtual font constructors. This method is for instance used for defining virtual math fonts.

The first method is what we use most. It is really important not to forget the feature specification. A rather safe bet is *default. In a next chapter we will discuss the difference between these two; here we focus on the name part.

The `filename` is in fact a symbolic name. In ConTEXt we have always used an indirect reference to fonts. Look at this:

```
\definefont[TitleFont][SerifBold*default sa 2]
```

A reference like `SerifBold` makes it possible to define styles independent of the chosen font family. This reference eventually gets resolved to a real name and there can be a chain of references.

Font definitions can be grouped into a larger setup using typescripts. In that case, we can set the features for a regular, italic, bold and bolditalic for the whole set but when a fontname has a specific feature associated (as in the previous examples) that one takes precedence.

so far we talked about fonts being files, but in practice a lookup happens by file as well as by name as known to the system. In the next section this will be explained in more detail.

3.3 File

You can force a file lookup with:

```
\definefont[TitleFont][file:somefilename*default sa 2]
```

If you use more symbolic names you can use the `file:` prefix in the mapping:

```
\definefontsynonym[SerifBold][file:somefile]
\definefont[TitleFont][SerifBold*default sa 2]
```

In projects that are supposed to run for a long time I always use the file based lookup, because filenames tend to be rather stable. Also, as the lookup happens in the TEX directory structure, file lookups will rely on the general file search routines. This has the benefit that case is ignored. When no match is found the lookup will also use the font name database. Spaces and special characters are ignored.

The name alone is not enough as there can be similar filenames with different suffixes. Therefore the lookup will happen in the order `otf`, `ttf`, `afm`, `tfm` and `lua`. You can force a lookup by being more explicit, like:

```
\definefont[TitleFont][file:somefilename.ttf*default sa 1]
```

3.4 Name

Say that we want to use a Dejavu font and that instead of filenames we want to use its given name. The best way to find out what is available is to call for a list:

```
mtxrun --script font --list --all dejavu
```

This produces the following list:

```
dejavusans                      dejavusans                      dejavusans.ttf
dejavusansbold                  dejavusansbold                  dejavusans-bold.ttf
dejavusansboldoblique           dejavusansboldoblique           dejavusans-boldoblique.ttf
dejavusanscondensed             dejavusanscondensed             dejavusanscondensed.ttf
dejavusanscondensedbold         dejavusanscondensedbold         dejavusanscondensed-bold.ttf
dejavusanscondensedboldoblique dejavusanscondensedboldoblique dejavusanscondensed-boldoblique.ttf
dejavusanscondensednormal       dejavusanscondensed             dejavusanscondensed.ttf
dejavusanscondensedoblique      dejavusanscondensedoblique      dejavusanscondensed-oblique.ttf
dejavusansextralight            dejavusansextralight            dejavusans-extralight.ttf
dejavusanslight                 dejavusansextralight            dejavusans-extralight.ttf
dejavusansmono                  dejavusansmono                  dejavusansmono.ttf
dejavusansmonobold              dejavusansmonobold              dejavusansmono-bold.ttf
dejavusansmonoboldoblique       dejavusansmonoboldoblique       dejavusansmono-boldoblique.ttf
dejavusansmononormal            dejavusansmonooblique           dejavusansmono-oblique.ttf
dejavusansmonooblique           dejavusansmonooblique           dejavusansmono-oblique.ttf
dejavusansnormal                dejavusans                      dejavusans.ttf
dejavusansoblique               dejavusansoblique               dejavusans-oblique.ttf
dejavuserif                     dejavuserif                     dejavuserif.ttf
dejavuserifbold                 dejavuserifbold                 dejavuserif-bold.ttf
dejavuserifbolditalic           dejavuserifbolditalic           dejavuserif-bolditalic.ttf
dejavuserifcondensed            dejavuserifcondensed            dejavuserifcondensed.ttf
dejavuserifcondensedbold        dejavuserifcondensedbold        dejavuserifcondensed-bold.ttf
dejavuserifcondensedbolditalic dejavuserifcondensedbolditalic dejavuserifcondensed-bolditalic.ttf
dejavuserifcondenseditalic      dejavuserifcondenseditalic      dejavuserifcondensed-italic.ttf
dejavuserifcondensednormal      dejavuserifcondensed            dejavuserifcondensed.ttf
dejavuserifitalic               dejavuserifitalic               dejavuserif-italic.ttf
dejavuserifnormal               dejavuserif                     dejavuserif.ttf
```

The first two columns mention the names that we can use to access a font. These are normalized names in the sense that we only kept letters and numbers. The next three definitions are equivalent:

```
\definefont[TitleFont][name:dejavuserif*default sa 1]
\definefont[TitleFont][name:dejavuserifnormal*default sa 1]
\definefont[TitleFont][name:dejavuserif.ttf*default sa 1]
```

In the list you see two names that all point to `dejavusans-extralight.ttf`:

```
dejavusansextralight
dejavusanslight
```

There are some heuristics built into ConTEXt and we do some cleanup as well. For instance we interpret `ital` as `italic`. In a font there is sometimes information about the weight and we look at those properties as well. Unfortunately font names (even within a collection) are often rather inconsistent so you still need to know what you're looking for. The more explicit you are, the less change of problems.

3.5 Spec

There is often some logic in naming fonts but it's not robust and really depends on how consistent a font designer or typefoundry has been. In ConTEXt we can access names by using a normalized scheme.

```
name-weight-style-width-variant
```

The following values are valid:

weight	`black bold demi demibold extrabold heavy light medium mediumbold normal regular semi semibold ultra ultrabold ultralight`
style	`italic normal oblique regular reverseitalic reverseoblique roman slanted`
width	`book condensed expanded normal thin`
variant	`normal oldstyle smallcaps`

The four specifiers are optional but the more you provide, the better the match. Let's give an example:

```
mtxrun --script fonts --list --spec dejavu
```

This reports:

```
dejavuserifcondensed normal normal normal normal dejavuserifcondensed dejavuserifcondensed.ttf
dejavuserif          normal normal normal normal dejavuserif          dejavuserif.ttf
dejavusansmono       normal normal normal normal dejavusansmono       dejavusansmono.ttf
dejavusanscondensed  normal normal normal normal dejavusanscondensed  dejavusanscondensed.ttf
dejavusans           normal normal normal normal dejavusans           dejavusans.ttf
```

We can be more specific, for instance:

```
mtxrun --script fonts --list --spec dejavu-bold
```

```
dejavuserif    bold normal normal normal dejavuserifbold     dejavuserif-bold.ttf
dejavusansmono bold normal normal normal dejavusansmonobold dejavusansmono-bold.ttf
dejavusans     bold normal normal normal dejavusansbold      dejavusans-bold.ttf
```

We add another specifier:

```
mtxrun --script fonts --list --spec dejavu-bold-italic
```

```
dejavuserif    bold italic normal normal dejavuserifbolditalic     dejavuserif-bolditalic.ttf
dejavusansmono bold italic normal normal dejavusansmonoboldoblique dejavusansmono-boldoblique.ttf
dejavusans     bold italic normal normal dejavusansboldoblique     dejavusans-boldoblique.ttf
```

As the first hit is used we need to be more specific with respect to the name, so lets do that in an example definition:

```
\definefont[TitleFont][spec:dejavuserif-bold-italic*default sa 1]
```

Watch the prefix spec. Wolfgang Schusters simplefonts module nowadays uses this method to define sets of fonts based on a name only specification. Of course that works best if a fontset has well defined properties.

4 Methods

4.1 Introduction

A font definition looks as follows:

```
\definefont
  [MyFont]
  [namepart method specification size]
```

For example:

```
\definefont
  [MyFont]
  [Bold*default at 12.3pt]
```

We have already discussed the namepart and size in a previous chapter and here we will focus on the method. The method is represented by a character and although we currently only have a few methods there can be many more.

4.2 : (direct features)

This one is seldom used, but those coming from another macro package to ConTEXt might use it as first attempt to defining a font.

```
\definefont
  [MyFont]
  [Bold:+kern;+liga; at 12.3pt]
```

This is the XƎTEX way of defining fonts. A + means as much as "turn on this feature" so you can guess what the minus sign does. Alternatively you can use a key/value approach with semicolons as separator. If no value is given the value yes is assumed.

```
\definefont
  [MyFont]
  [Bold:kern=yes;liga=yes; at 12.3pt]
```

When we started supporting XƎTEX we ran into issues with already present features of ConTEXt as the XƎTEX syntax also has some more obscure properties using slashes and brackets for signalling a file or name lookup. As in ConTEXt we prefer a more symbolic approach anyway, it never was a real issue.

4.3 * (symbolic features)

The most natural way to associate a set of features with a font instance is the following:

```
\definefont
  [MyFont]
  [Bold*default at 12.3pt]
```

This will use the featureset named default and this one is defined in font-pre.mkiv which might be worth looking at.

```
\definefontfeature
  [always]
  [mode=auto,
   script=auto,
   kern=yes,
   mark=yes,
   mkmk=yes,
   curs=yes]

\definefontfeature
  [default]
  [always]
  [liga=yes,
   tlig=yes,
   trep=yes] % texligatures=yes,texquotes=yes

\definefontfeature
  [smallcaps]
  [always]
  [smcp=yes,
   tlig=yes,
   trep=yes] % texligatures=yes,texquotes=yes

\definefontfeature
  [oldstyle]
  [always]
  [onum=yes,
   liga=yes,
   tlig=yes,
   trep=yes] % texligatures=yes,texquotes=yes

\definefontfeature % == default unless redefined
  [ligatures]
  [always]
  [liga=yes,
   tlig=yes,
   trep=yes]

\definefontfeature % can be used for type1 fonts
```

Methods

```
  [complete]
  [always]
  [compose=yes,
   liga=yes,
   tlig=yes,
   trep=yes]

\definefontfeature
  [none]
  [mode=none,
   features=no]
```

These definitions show that you can construct feature sets on top of existing ones, but keep in mind that they are defined instantly, so any change in the parent is not reflected in its kids.

In a font definition you can specify more than one set:

```
\definefont
  [MyFont]
  [Bold*always,oldstyle at 12.3pt]
```

4.4 @ (virtual features)

This method is somewhat special as it demands knowledge of the internals of the Con-TEXt font code. Much of it is still experimental but it is a nice playground. A good example of its usage can be found in the file m-punk.mkiv where we create a font out of MetaPost graphics.

Another example is virtual math. As in the beginning of LuaTEX and MkIV there were only a few OpenType math fonts, and as I wanted to get rid of the old mechanisms, it was decided to virtualize the math fonts. For instance a Latin Modern Roman 10 point math font can be defined as follows:

```
\definefontsynonym
  [LMMathRoman10-Regular]
  [LMMath10-Regular@lmroman10-math]
```

The lmroman10-math refers to a virtual definition and in this case it is one using a built-in constructor and therefore we use a goodies file to specify the font. That file looks as follows:

```
return {
  name = "lm-math",
  version = "1.00",
  comment = "Goodies that complement latin modern math.",
```

```
      author = "Hans Hagen",
      copyright = "ConTeXt development team",
      mathematics = {
        ...
        virtuals = {
          ...
          ["lmroman10-math"] = ten,
          ...
        },
        ...
      }
    }
```

Here `ten` is a previously defined table:

```
local ten = {
  { name = "lmroman10-regular.otf", features = "virtualmath", main = true },
  { name = "rm-lmr10.tfm", vector = "tex-mr-missing" } ,
  { name = "lmmi10.tfm", vector = "tex-mi", skewchar = 0x7F },
  { name = "lmmi10.tfm", vector = "tex-it", skewchar = 0x7F },
  { name = "lmsy10.tfm", vector = "tex-sy", skewchar = 0x30, parameters = true } ,
  { name = "lmex10.tfm", vector = "tex-ex", extension = true } ,
  { name = "msam10.tfm", vector = "tex-ma" },
  { name = "msbm10.tfm", vector = "tex-mb" },
  { name = "stmary10.afm", vector = "tex-mc" },
  { name = "lmroman10-bold.otf", vector = "tex-bf" } ,
  { name = "lmmib10.tfm", vector = "tex-bi", skewchar = 0x7F } ,
  { name = "lmsans10-regular.otf", vector = "tex-ss", optional = true },
  { name = "lmmono10-regular.otf", vector = "tex-tt", optional = true },
  { name = "eufm10.tfm", vector = "tex-fraktur", optional = true },
  { name = "eufb10.tfm", vector = "tex-fraktur-bold", optional = true },
}
```

This says as much as: take `lmroman10-regular.otf` as starting point and overload slots with ones found in the following fonts. The vectors are predefined as they are shared with other font sets like `px` and `tx`.

In due time more virtual methods might end up in ConTeXt because they are a convenient way to extend or manipulate fonts.

4.5 Lua fonts

You can define a font in Lua. In the process you can use all kind of helper functions that ConTeXt provides. Here is an example:

```
local startactualtext = backends.codeinjections.startunicodetoactualtext
```

```
local stopactualtext  = backends.codeinjections.stopunicodetoactualtext

return function(specification)
    local features = specification.features.normal
    local name     = features.original or "dejavu-serif"
    local option   = features.option       -- we only support "line"
    local size     = specification.size   -- always set
    local detail   = specification.detail -- e.g. default
    if detail then
        name = name .. "*" .. detail
    end
    local f, id = fonts.constructors.readanddefine(name,size)
    if f then
        f.properties.name = specification.name
        f.properties.virtualized = true
        f.fonts = {
            { id = id },
        }
        for s in string.gmatch("aeuioy",".") do
            local n = utf.byte(s)
            local c = f.characters[n]
            if c then
                local w = c.width  or 0
                local h = c.height or 0
                local d = c.depth  or 0
                if option == "line" then
                    f.characters[n].commands = {
                        { "special", "pdf:direct:" .. startactualtext(n) },
                        { "rule", option == "line" and size/10, w },
                        { "special", "pdf:direct:" .. stopactualtext() },
                    }
                else
                    f.characters[n].commands = {
                        { "special", "pdf:direct:" .. startactualtext(n) },
                        { "down", d },
                        { "rule", h + d, w },
                        { "special", "pdf:direct:" .. stopactualtext() },
                    }
                end
            else
                -- probably a real bad font
            end
        end
    end
    return f
```

```
end
```

This code is stored in `fonts-demo-rule.lua` and we can load that font in the usual way, by specifying a filename:

```
\definefont
  [MyRuleFont]
  [file:fonts-demo-rule.lua*default sa 1]
```

S█ wh█n w█ █s█ █t w█ g█t t█xt t█p█s█t wh█n█ █ll v█w█ls █r█ r█pl█c█d b█ r█l█s. Th█ █ct██lt█xt █nj█ct██n (█n th█ ██ m█) m█k█s █t p█ss█bl█ t█ c█t █nd p█st█ th█ t█xt fr█m th█ pdf d█c█m█nt b█t wh█l█ wr█t█ng th█s (m█d 2016) █ mupdf b█s█d v██w█r c██ldn't h█ndl█ █t █nd acrobat h█d pr█bl█ms w█th sp█c█s.

```
\definefontfeature
  [myrulefont]
  [default]
  [original=file:texgyrepagella-regular.otf]
\definefont
  [MyRuleFont]
  [file:fonts-demo-rule.lua*myrulefont]
```

Th█ pr█v███s c█d█ d█m█nstr█t█s h█w w█ c█n p█ss █ f█ntn█m█ t█ b█ █s█d █s b█s█ t█ th█ g█n█r█t█r. In c█s█ ███ w█nd█r h█w f██t█r█s b█h█v█ w█th s█ch f█nts: █s ███ c█n s██ h█r█, f█nt k█rns █r█ █nd██d █nj█ct█d. C█mp█r█d t█ Dejavu, th█ Pagella f█nt h█s q██t█ s█m█ m█r█ k█rns.

```
\definefontfeature
  [myrulefont]
  [default]
  [original=file:texgyrepagella-regular.otf,
   option=line]
\definefont
  [MyRuleFont]
  [file:fonts-demo-rule.lua*myrulefont]
```

H█r█ w█ sh█w h█w th█ p█ss█d option █s h█ndl█d. B█c██s█ w█ n█ l█ng█r h█v█ █ r█l█t█nsh█p w█th th█ h██ght █nd d█pth, th█ r██l t█xt █s █ b█t h█rd█r t█ g██ss.

4.6 Old fuzzy fonts

Most natural is to use OpenType or Type1 fonts. In the case of Type1 a matching pair of `afm` and `pfb` files is needed. However, there can be situations where there is only a `tfm` and `pfb` file (or not even that: just a bitmap file).

I will not show specimen here, simply because I don't have (nor want to have) the fonts needed in my development and production environments. The implementation was tested with a specific czech computer modern font.

In a traditional (8 bit) setup we have an `tfm` file, a `pfb` file and a `enc file`. The order of the characters in the `tfm` file directly relates to the input encoding. The `enc` file relates that order to the order in the `pfb` file. The mapping from input encoding to font shape encoding happens via glyph names. In the `map` file we tell what `pfb` file to use with what `enc` file.

However, in the case of the `csr.tfm` and `csr.pfb` file it looks like in practice the `enc` file is not used, probably because in the `pfb` file the standard encoding matches the order in the `tfm` file. This is of course a rather dangerous assumption, especially if information lacks to check it.

The next example definitions demonstrate several paths to go from Unicode input (source file) to rendered shapes. As this is mostly meant for generic usage we use the low level definition code (ConTEXt users are not supposed to use that method).

```
\font\foo=file:csr10.tfm:reencode=auto;mode=node;liga=yes;kern=yes
```

This is the easiest way. We use the `tfm` file for dimensions, ligatures and kerns. The `auto` option will use the `pfb` file to identify the right mapping. We enable ligatures and kerns and we use node mode. This indicates that we're dealing with a pseudo OpenType setup here. You can provide a `pfb` file with the `pfbfile` feature in case the name differs from the `tfm` file.

```
\font\foo=file:csr10.tfm:reencode=csr.enc;mode=node;liga=yes;kern=yes
```

Now we use the `enc` file for the encoding vector but we still need the `pfb` file for mapping that onto the right shape. You probably can best use `auto` instead.

```
\font\foo=file:csr10.tfm:reencode=csr.enc;bitmap=yes;mode=node;liga=yes;kern=yes
```

Here we force bitmap shapes. This is a bit tricky as a different code path is followed in the backend. Unless the situation is too confusing, a proper `ToUnicode` is included in the output, so that cut and paste works all right, given that the viewer is able to deal with it (always use Acrobat as reference).

Why do we need modes and/or to simulate OpenType behaviour? Indeed it seldom makes sense with `tfm` files but in this particular case teh font has a quote cheat.

```
\startluacode
    fonts.handlers.otf.addfeature {
        name = "czechdqcheat",
        type = "substitution",
        data = {
            quotedblright = "csquotedblright",
        },
    }
\stopluacode
```

We could make this a language specific feature but as this font is not meant for other languages it makes no sense to do so. This feature is enabled with:

```
czechdqcheat=yes
```

This will replace one quote by another with different side bearings. Of course a properly bounded quote with proper kerning makes much more sense. A test case is:

```
\quotedblleft  X\quotedblright
\quotedblright X\quotedblleft
```

5 Features

5.1 Introduction

If you look into fonts, it is hard not to bump into kerns (spacing between characters) and ligatures (combining multiple shapes into one) and apart from monospaced fonts most Type1 fonts have them. In the OpenType universe we call these properties features and in such a font there can be many such features.

For those who grew up with TEX or still remember the times of eight bit fonts, it is no secret that TEX macro packages did some magic to get most out of a font: replacing missing glyphs, fixing metrics, using commands to access shapes that had a weird code point, to mention a few. As there is absolutely no guarantee that an OpenType font does better, there is a good reason to continue messing around with fonts. After all, it's what TEX users seem to like: control.

So, when we started writing support for OpenType quite soon a mechanism has been created that permits adding our own features to the repertoire that comes with a font. Because OpenType features demand a configuration and control mechanism, it made sense to generalize that and provide a single interface.

This means that when we talk about font features, we don't limit ourselves to those provided by the font, but also those provided by ConTEXt. As with font features, they are enabled per font.

Some of the extra features are sort of generic, others are very font specific and their properties are somewhat bound to a font. Such features are defined in a font goodie files. Consider these goodies a font extension mechanism.

Some features need information that only the engine can provide. This is why we have analyzers. Some are generic, others are bound to scripts. They come in action before features are applied. Rather special is applying features in combination with paragraph building. This is something very specific to ConTEXt but it depends on properties of the font. It falls into the category 'optimizing'.

It is clear that when we talk of features many aspects of a font play a role. In this chapter we will discuss all the mentioned aspects. There is quite a bit of Lua code shown in this chapter, but don't worry, users will seldom need to tweak fonts this way. On the other hand it's good to see what is possible.

5.2 Regulars

5.2.1 Introduction

The OpenType specification, which can be found on the Microsoft website is no easy reading. Some of the concepts are easy to understand, like relative positioning (that we

call kerning in TEX) or ligature substitution (as we have ligatures in TEX too). It makes no sense to discuss the bitwise composition of an OpenType or TrueType file here. First of all, all we get to see is a Lua table, and in ConTEXt even that one gets sanitized and optimized into a more useable table. However, as the data that comes with a font is a good indication of what a font is capable of, we will discuss some of it in an appendix. In this section we will discuss the basic principles and categories of features.

5.2.2 Feature sets

Because in the next examples we will demonstrate features, we need to know how we can tell ConTEXt what features to use. Although you can add explicit feature definitions to a font specification, I strongly advice you not to do this but use the more abstract mechanism of feature sets. These are defined as follows:

```
\definefontfeature
  [MyFeatureSet]
  [alpha=yes,
   beta=no,
   gamma=123]
```

Such a set is bound to a font with the * specifier, as in:

```
\definefont
  [MyFontInstance]
  [MyNiceFont*MyFeatureSet at 12pt]
```

In most cases the already defined default feature set will suffice. It often makes sense to use that one as base for new definitions:

```
\definefontfeature
  [MyFeatureSet]
  [default]
  [alpha=yes,
   beta=no,
   gamma=123]
```

The second argument can be a list, as in:

```
\definefontfeature
  [MyFeatureSet]
  [MyFirstSet,MySecondSet]
  [alpha=yes,
   beta=no,
   gamma=123]
```

Of course you need to know what features a font support, and one way to find out is:

```
mtxrun --script font --list --info --pattern=pagella
```

Don't be too surprised if different fonts show different features and even similar features can be implemented differently. Sometimes you really need to know the font, but fortunately many fonts come with examples.

There are many features and there values are kept with the font when it gets defined. This means that when you change a featureset, it will not affect already defined fonts. Because fonts are often defined on demand, you need to be aware of the fact that a redefinition of a featureset can have consequences for already defined fonts. For instance, a bodyfont switch only sets up the fonts and delays defining them.

Although features are a sort of abstractions it can be interesting to see what features and values are actually used:

```
\usemodule[fonts-features] \showusedfeatures
```

You will notice that we have more features than OpenType fonts can offer. This is because in ConTEXt features is a more general concept.

feature		description	value
aalt		access all alternates	yes
abvm		above-base mark positioning	yes
abvs		above-base substitutions	yes
akhn		akhands	yes
analyze	+	analysis of character classes	yes
anum	+	arabic digits	yes
autolanguage	-		position
autoscript	-		position
blwf		below-base forms	yes
blwm		below-base mark positioning	yes
blws		below-base substitutions	yes
calt		contextual alternates	yes
ccmp		glyph composition/decomposition	yes
char-ligatures	+	unicode char specials to ligatures	yes
cjct		conjunct forms	yes
clig		contextual ligatures	no
			yes
cmcp	-		yes
compat-ligatures	+	unicode compat specials to ligatures	yes
compose	+	additional composed characters	yes
curs		cursive positioning	yes
dist		distances	yes
dlig		discretionary ligatures	no
			yes
expansion	+	apply hz optimization	quality
extend	+	scale glyphs horizontally	.8
			1.2

```
features         + features                           no
fin2               terminal forms #2                 yes
fin3               terminal forms #3                 yes
fina               terminal forms                    yes
goodies          + goodies on top of built in features  dingbats
half               half forms                        yes
haln               halant forms                      no
                                                     yes
init               initial forms                     yes
isol               isolated forms                    yes
js16             -                                   yes
keepligatures    + keep ligatures in letterspacing   auto
kern               kerning                           yes
lang             -                                   japanese
liga               standard ligatures                no
                                                     yes
ljmo               leading jamo forms                yes
lnum               lining figures                    no
locl               localized forms                   yes
mark               mark positioning                  yes
mathalternates   + additional math alternative shapes  yes
mathitalics      + additional math italic corrections  yes
mathnolimitsmode + influence nolimits placement      0,800
mathsize         + apply mathsize specified in the font  yes
med2               medial forms #2                   yes
medi               medial forms                      yes
missing          + missing symbols                   yes
mkmk               mark to mark positioning          yes
mode             + mode                              base
                                                     node
                                                     none
nostackmath      + disable math stacking mechanism   yes
nukt               nukta forms                       yes
onum               old style figures                 no
                                                     yes
option           -                                   line
original         -                                   file:texgyrepagella-regula
pnum               proportional figures              no
                                                     yes
pref               pre-base forms                    yes
pres               pre-base substitutions            yes
protrusion       + l/r margin character protrusion   quality
pstf               post-base forms                   yes
psts               post-base substitutions           yes
```

rkrf		rakar forms	yes
rlig		required ligatures	no
			yes
rphf		reph form	yes
rtlm		right to left math	yes
salt		stylistic alternates	yes
slant	+	slant glyphs	.2
smcp		small capitals	no
			yes
ss01	-		yes
ss03	-		yes
ss10	-		yes
ss12	-		yes
ss15	-		yes
ss16	-		yes
ss19	-		yes
ss24	-		yes
ss25	-		yes
ss26	-		yes
ss27	-		yes
ss31	-		yes
ss34	-		yes
ss35	-		yes
ss36	-		yes
ss37	-		yes
ss38	-		yes
ss41	-		yes
ss42	-		yes
ss43	-		yes
ss60	-		yes
ssty		script style	1
			2
			no
sups		superscript	yes
tjmo		trailing jamo forms	yes
tlig	+	tex ligatures	yes
tnum		tabular figures	no
			yes
trep	+	tex replacements	yes
unicoding	+	adapt unicode table	yes
vatu		vattu variants	yes
vjmo		vowel jamo forms	yes
zero		slashed zero	yes

5.2.3 Main categories

There are two (but potentially more) main groups of features: those that deal with substitution, and those that lead to positioning. It is not really needed to know the gory details, but it helps to know at least a bit about them as it can help to track down issues with fonts.

There are several substitutions possible:

- a single substitution replaces one glyph by another
- a multiple substitution replaces one glyph by one or more
- a ligature substitution replaces multiple glyphs by one glyph
- an alternate substitution replaces one glyph by one out of a set

Like it or not, but these categories are not always used as intended: they just are a way of replacing one or more glyphs by one or more other glyphs. This means that when for instance ij gets replaced by one glyph (given that the font supports it) a ligature substitution is used, even when in fact we have to do with a diftong that can be represented by one character.

No matter what features you will use, keep in mind that they are nothing more than a combination of substitutions and positioning directives. So, the de facto standard ligature building feature liga indeed uses a ligature substitution, but other features with names that resemble no ligatures might use this substitution as well.

An example of a single substitution is an oldstyle (onum) although it can as well be implemented as a choice out of alternate glyphs. Another example is smallcaps (smcp). Nowdays these are more or less standard features for a grown up font, while in the past they came as separate fonts. So, instead of loading an extra font, one sticks to one and selects the feature that does the substitution.

A second category concerns relative positioning. Again we have several variants:

- a single positioning moves a glyph over one of two axis and can change the width and/or height
- a pair positioning also moved glyphs but concerns two adjacent glyphs
- a cursive positioning operates on a range of glyphs and is used to visually connect them

In addition there are three ways to anchor marks onto glyphs:

- a mark can be anchored on a base glyph
- a mark can be anchored on a specific (visual) component of a ligature
- a mark can be anchored on another mark

In base mode the single, alternate and ligature substitutions can rather easily be mapped onto the traditional TeX font handling mechanism and this is what happens in base mode. A single substitution is just another instance of a glyph so there we just replace the

original index into the glyph table by another one. In the case of an alternate we change the default index into one of several possible replacements in the alternate set. Ligatures can be mapped onto TEXs ligature mechanism. The single positioning maps nicely on TEXs kerning mechanism and pairwise positioning is not applicable in base mode. In node mode we don't do any remapping at loading time but delegate that to Lua when processing the node lists.

Marks are special in the sense that they normally only occur in scripts that also use substitution and positioning which in turn means that some more housekeeping is involved. After all, we need to keep track to what a mark applies. Of course a font can provide regular latin accents as marks but that is ill practice because cut and paste might not work out as expected. A proper font will support composed characters and provide glyphs that have the accents built in. Marks are not dealt with in base mode.

Talking of complex scripts, the above set of operations is far from enough. Take for instance Arabic, where a sequence of 5 characters with 3 marks can easily become two glyphs glued together with two marks only. In the process we can have single substitutions, ligatures being built, marks being anchored and glyphs being cursively positioned. But, in order to do this well, some contextual analysis has to be done as well. Again we have several variants of this:

- with contextual substitution a replacement takes place depending on a matching sequence of glyphs, optionally preceded or followed by matches
- with contextual positioning shifting and anchoring happens based on a matching sequence of glyphs, optionally preceded or followed by matches
- multiple contextual substitutions or positionings can be chained together
- this can also happen in the reverse order (for right-to-left scripts)

In practice there is no fundamental difference between these and we can collapse them all in a sequence of lookups resulting in a sequence of whatever other manipulation is wanted.

Given this, what is a feature? It's mostly a sequence of actions expressed in the above. And although there is a whole repertoire of semi-standardized features like `liga` and `onum`, there is no real hard coded support for them in ConTEXt. Instead we have a generic feature processor that deals with all of them. A feature, say `abcd`, has a definition that boils down to a sequence of lookups. A lookup is just a name that is associated to one of the mentioned actions. So, `abcd` can do a decomposition (multiple substitution), then a replacement (single substitution) based on neighbouring glyphs, then do some ligature building (ligature substitution) and finally position the resulting glyphs relative to each other (like cursive positioning and anchoring marks).

Imagine that we start out with 5 characters in the input. Instead of real glyphs we represent them by rectangles. The third one is a mark.
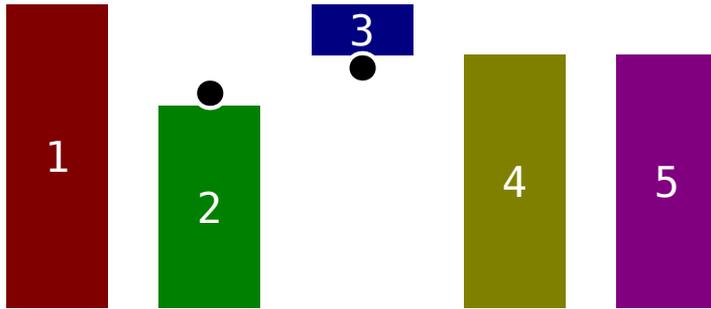
In the next variant we see that four and five have been replaced by number six. This is a ligature replacement.
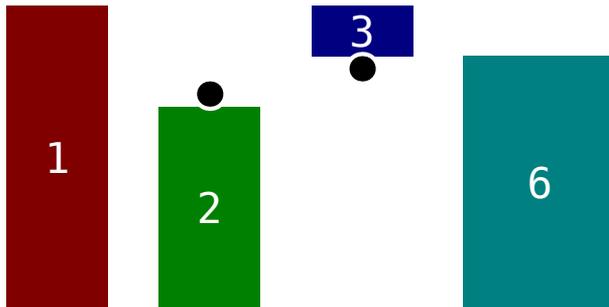
The mark is an independent entity. Sometimes it has a width, sometimes it hasn't. In both cases we can position it. Here we move the shape left and down. There are two ways to do this: simple pairwise kerning but better is to use anchors. Here we have one anchor per shape but there can be many.
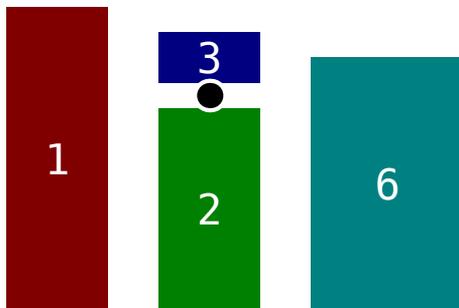
Next we apply some kerning. Of course the anchored marks need to move as well.
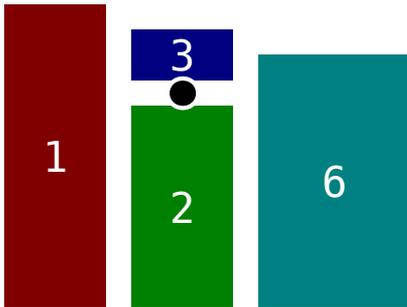
Alternatively we can connect the shapes in a cursive way. The name cursive is somewhat misleading as it just boils down to shifting. The cursive indicates that the shifts accumulate within a word.
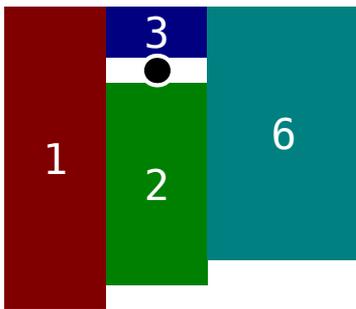
5.2.4 Single substitution

Single substitutions are probably the most used ones. For instance, when you ask for small caps, a lot of glyphs get replaced. When using oldstyle numerals only digits get replaced but even then each glyph has to be checked. This can be demonstrated with the Latin Modern fonts.

$123.45 $123.45

As you can see here, Latin Modern has an oldstyle dollar sign. If you don't like that one, you're in troubles as it comes with the rest of the oldstyles. The only way out is to apply the oldstyle numerals to digits only which involves more tagging than you might be willing to add. So, whenever you choose a substitution, be aware that you have not that much control over what gets substituted: it's the font that drives it. Here are some examples:

```
\definefontfeature[capsandold][smallcaps,oldstyle]

\showotfcomposition{dejavu-serif*capsandold      at 24pt}{}{It's 2013!}
\showotfcomposition{cambria*capsandold           at 24pt}{}{It's 2013!}
\showotfcomposition{lmroman10regular*capsandold  at 24pt}{}{It's 2013!}
```

\showotfcomposition{texgyrepagellaregular*capsandold at 24pt}{}{It's 2013!}

font 65: DejaVuSerif.ttf @ 24.0pt

features analyze=yes, autolanguage=position, autoscript=position,
checkmarks=yes, curs=yes, devanagari=yes, dummies=yes,
extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes,
liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes,
mode=node, onum=yes, script=dflt, smcp=yes, spacekern=yes,
tlig=yes, trep=yes

step 1 It's 2013! U+49:I U+74:t U+27:' U+73:S [glue]
U+32:2 U+30:0 U+31:1 U+33:3 U+21:!

feature 'trep', type 'gsub_single', lookup 'trep', replacing
U+00027 (quotesingle) by single U+02019 (quoteright)

result It's 2013! U+49:I U+74:t U+2019:' U+73:S [glue]
U+32:2 U+30:0 U+31:1 U+33:3 U+21:!
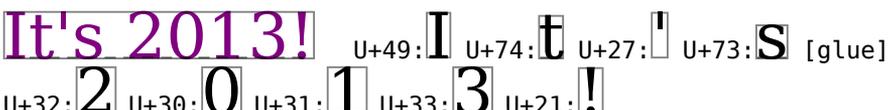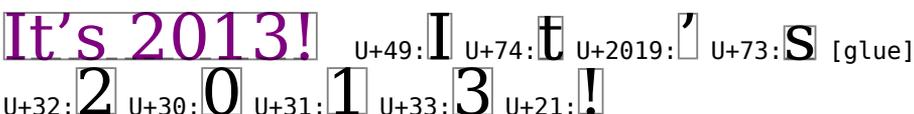
font 66: cambria.ttf @ 24.0pt

features analyze=yes, autolanguage=position, autoscript=position,
checkmarks=yes, curs=yes, devanagari=yes, dummies=yes,
extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes,
liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes,
mode=node, onum=yes, script=latn, smcp=yes, spacekern=yes,
tlig=yes, trep=yes
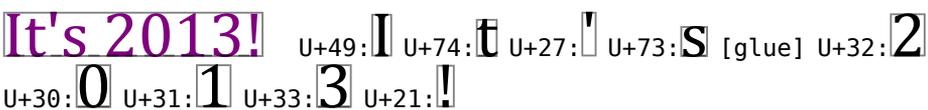
step 1 It's 2013! U+49:I U+74:t U+27:' U+73:S [glue] U+32:2
U+30:0 U+31:1 U+33:3 U+21:!

feature 'trep', type 'gsub_single', lookup 'trep', replacing
U+00027 by single U+02019

step 2 It's 2013! U+49:I U+74:t U+2019:' U+73:S [glue]
U+32:2 U+30:0 U+31:1 U+33:3 U+21:!
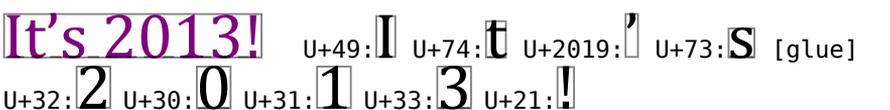
feature 'smcp', type 'gsub_single', lookup 's_s_7', replacing
U+00074 by single U+F0015

feature 'smcp', type 'gsub_single', lookup 's_s_7', replacing
U+00073 by single U+F0014

step 3 IT's 2013! U+49:I U+F0015:T U+2019:' U+F0014:S
[glue] U+32:2 U+30:0 U+31:1 U+33:3 U+21:!
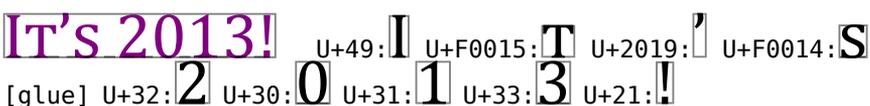
```
            feature 'onum', type 'gsub_single', lookup 's_s_23', replacing
              U+00032 by single U+F0147

            feature 'onum', type 'gsub_single', lookup 's_s_23', replacing
              U+00030 by single U+F0145

            feature 'onum', type 'gsub_single', lookup 's_s_23', replacing
              U+00031 by single U+F0146

            feature 'onum', type 'gsub_single', lookup 's_s_23', replacing
              U+00033 by single U+F0148
```

result IT'S 2013! U+49:I U+F0015:T U+2019:' U+F0014:S [glue] U+F0147:2 U+F0145:0 U+F0146:1 U+F0148:3 U+21:!

font 67: lmroman10-regular.otf @ 24.0pt

features analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes, mode=node, onum=yes, script=dflt, smcp=yes, spacekern=yes, tlig=yes, trep=yes

step 1 It's 2013! U+49:I U+74:t U+27:' U+73:S [glue] U+32:2 U+30:0 U+31:1 U+33:3 U+21:!

```
            feature 'trep', type 'gsub_single', lookup 'trep', replacing
              U+00027 (quotesingle) by single U+02019 (quoteright)
```

step 2 It's 2013! U+49:I U+74:t U+2019:' U+73:S [glue] U+32:2 U+30:0 U+31:1 U+33:3 U+21:!

```
            feature 'onum', type 'gsub_single', lookup 's_s_4', replacing
              U+00032 (two) by single U+0F732 (two.oldstyle)

            feature 'onum', type 'gsub_single', lookup 's_s_4', replacing
              U+00030 (zero) by single U+0F730 (zero.oldstyle)

            feature 'onum', type 'gsub_single', lookup 's_s_4', replacing
              U+00031 (one) by single U+0F731 (one.oldstyle)

            feature 'onum', type 'gsub_single', lookup 's_s_4', replacing
              U+00033 (three) by single U+0F733 (three.oldstyle)
```

result It's 2013! U+49:I U+74:t U+2019:' U+73:S [glue] U+F732:2 U+F730:0 U+F731:1 U+F733:3 U+21:!

font 68: texgyrepagella-regular.otf @ 24.0pt

features analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes, mode=node, onum=yes, script=dflt, smcp=yes, spacekern=yes, tlig=yes, trep=yes

step 1 It's 2013! U+49:I U+74:t U+27:' U+73:S [glue] U+32:2 U+30:0 U+31:1 U+33:3 U+21:!

  feature 'trep', type 'gsub_single', lookup 'trep', replacing U+00027 (quotesingle) by single U+02019 (quoteright)

step 2 It's 2013! U+49:I U+74:t U+2019:' U+73:S [glue] U+32:2 U+30:0 U+31:1 U+33:3 U+21:!

  feature 'smcp', type 'gsub_single', lookup 's_s_3', replacing U+00074 (t) by single U+0F774 (t.sc)

  feature 'smcp', type 'gsub_single', lookup 's_s_3', replacing U+00073 (s) by single U+0F773 (s.sc)

step 3 It's 2013! U+49:I U+F774:T U+2019:' U+F773:S [glue] U+32:2 U+30:0 U+31:1 U+33:3 U+21:!

  feature 'onum', type 'gsub_single', lookup 's_s_5', replacing U+00032 (two) by single U+0F732 (two.oldstyle)

  feature 'onum', type 'gsub_single', lookup 's_s_5', replacing U+00030 (zero) by single U+0F730 (zero.oldstyle)

  feature 'onum', type 'gsub_single', lookup 's_s_5', replacing U+00031 (one) by single U+0F731 (one.oldstyle)

  feature 'onum', type 'gsub_single', lookup 's_s_5', replacing U+00033 (three) by single U+0F733 (three.oldstyle)

result It's 2013! U+49:I U+F774:T U+2019:' U+F773:S [glue] U+F732:2 U+F730:0 U+F731:1 U+F733:3 U+21:!

Features

5.2.5 Multiple substitution

In a multiple substitution a sequence of characters (glyphs) gets replaced by another sequence. In fact, you might wonder why one–to–one, multiple–to–one and multiple–to–multiple are not all generalized into this variant. Efficiency is probably the main reason.[6] For instance the many–to–one is often used for ligatures (`liga`) and as a consequence `liga` is often misused also for non–ligatures.

One usage of multiple replacements is to avoid and or undo other replacements. In the next example we see a language dependent `fi` ligature. Take the dutch `ij` and `ie` diftongs. Here we need to prevent the `i` becoming combined with the `f` as it would look weird. Among the solutions for this are: context dependent ligatures (which involves a lot of rules), or multiple to multiple replacements (looking at the `fij` sequence).

```
\definefontfeature[default-fijn-en][default][language=eng,script=latn]
\definefontfeature[default-fijn-nl][default][language=nld,script=latn]

\definedfont[lmroman10-regular*default-fijn-en]\en effe fijn fietsen
\definedfont[lmroman10-regular*default-fijn-nl]\nl effe fijn fietsen
```

This gives:

effe fijn fietsen
effe fijn fietsen

Of course from this result one cannot see what (combination of) substitution(s) was used, but it's a nice exercise to work out a solution.

Multiple substitutions are mostly used for scripts more complex than latin or special fonts like Zapfino where advanced contextual analysis happens.

5.2.6 Alternate substitution

Alternates are simple one–to–one substitutions. Popular examples are small capitals and oldstyle numerals.

A nice application of alternates is the punk font. This font is a Knuth original. As part of experimenting with the MetaPost library in the early days of LuaTeX and MkIV, run-time randomization was implemented. However, that variant used virtual fonts and was

---

[6] Isn't it strange that complex mechanisms are designed to save a few bytes while at the same time we produce ridiculous large pictures with cameras.

somewhat resource hungry. So, in a later stage Khaled Hosny made an OpenType version using MetaPost output. Randomization is implemented through the `rand` feature.

In MkIV the `rand` feature is not really special and behaves just like any other (stylistic) alternate. The only difference is that for this feature a value of `yes` equals `random`. This also means that any feature that uses alternates use them randomly.

```
\definefontfeature[punknova-first] [mode=node,kern=yes,rand=first]
\definefontfeature[punknova-2]      [mode=node,kern=yes,rand=2]
\definefontfeature[punknova-yes]    [mode=node,kern=yes,rand=yes]
\definefontfeature[punknova-random][mode=node,kern=yes,rand=random]
```

We use this is:

```
The original punk font is designed by Don Knuth: xxxxxxxxxxxx
```

```
\definedfont[punknova-regular                  at 15pt] \getbuffer[sample]
\definedfont[punknova-regular*punknova-first  at 15pt] \getbuffer[sample]
\definedfont[punknova-regular*punknova-2       at 15pt] \getbuffer[sample]
\definedfont[punknova-regular*punknova-yes    at 15pt] \getbuffer[sample]
\definedfont[punknova-regular*punknova-random at 15pt] \getbuffer[sample]
```

In order to illustrate the variants we show a sequence of x's. There are upto ten different variants per characters.

There is one pitfall with random alternates: if each run leads to a different outcome, we can end up in oscillation: different shapes give different paragraphs and we can get more pages or cross references etc. that can end up differently so this is why ConTEXt always uses the same random seed (which gets reset when you purge the auxiliary files.

5.2.7 Ligature substitution

A ligature is traditionally a combination of several characters into one. Popular ligatures are 'fi', 'fl', 'ffi' and , 'ffl'. Occasionally we see 'æ', 'œ' and some more. Often ligatures are language dependant. For instance in languages like Dutch and German there can be compound words where one part ends with an f and the next part starts with an `f` and that looks bad or at least not intuitive. To some extent one can wonder if this tradition of ligatures is a good one. It definitely made sense ages ago, but I wouldn't be surprised if they are often added to fonts because the encoding vectors have them. After all, nothing prevents to go ahead and come up with way more ligatures.

There can be many ligature features in a font. Although we support arbitrary features, that is: those not registered as being official one way or the other, the following are known by description:

clig   contextual ligatures
dlig   discretionary ligatures
hlig   historical ligatures
liga   standard ligatures
rlig   required ligatures
tlig   traditional tex ligatures

The `default` feature set has type liga as wel as the T_EX specific `tlig` that replaces successive hyphen signs into en- and emdashes. The `arabic` feature set also has `rlig` enabled.

Now, there is one thing you should realize when we discuss specific features and the underlaying mechanisms: there is no real relationship between the features's name and the mechanisms used: any feature can use any underlying mechanism or combination. This is why deep down we see that what is internally called ligature gets used for any purpose where multiple–to–one replacements happen, and why the `liga` feature can use single substitutions or alternates to swap in another rendering so that the dot of the `i` stays free of the preceding `f`. And for some fonts relative positioning can be used to achieve a ligature effect.

The next examples demonstrate how the `liga` feature deals with `ffi`. Possible solutions are: replace all three at once, replace the first two first and in a next step, combine a ligature and following character, replace one or more components by variants that have no interference with the dot of the 'i'.

```
\showotfcomposition{dejavu-serif*default         at 48pt}{}{ffi}
\showotfcomposition{cambria*default              at 48pt}{}{ffi}
\showotfcomposition{lmroman10regular*default     at 48pt}{}{ffi}
\showotfcomposition{texgyrepagellaregular*default at 48pt}{}{ffi}
```

font   75: DejaVuSerif.ttf @ 48.0pt

features analyze=yes, autolanguage=position, autoscript=position,
              checkmarks=yes, curs=yes, devanagari=yes, dummies=yes,
              extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes,
              liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes,
              mode=node, script=dflt, spacekern=yes, tlig=yes, trep=yes

step 1

              feature 'liga', type 'gsub_ligature', lookup 's_s_3', replacing
              U+00066 (f) upto U+00066 (f) by ligature U+0FB00 case 2

result   ffi U+FB00: ff U+69: i

font   76: cambria.ttf @ 48.0pt

features   analyze=yes, autolanguage=position, autoscript=position,
checkmarks=yes, curs=yes, devanagari=yes, dummies=yes,
extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes,
liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes,
mode=node, script=latn, spacekern=yes, tlig=yes, trep=yes

step 1   ffi U+66: f U+66: f U+69: i

feature 'liga', type 'gsub_contextchain', chain lookup
's_s_17', index 0, replacing single U+00066 by U+F016D

result   ffi U+66: f U+F016D: f U+69: i

font   77: lmroman10-regular.otf @ 48.0pt

features   analyze=yes, autolanguage=position, autoscript=position,
checkmarks=yes, curs=yes, devanagari=yes, dummies=yes,
extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes,
liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes,
mode=node, script=dflt, spacekern=yes, tlig=yes, trep=yes

step 1   ffi U+66: f U+66: f U+69: i

feature 'liga', type 'gsub_ligature', lookup 's_s_8', replacing
U+00066 (f) upto U+00066 (f) by ligature U+0FB00 (f_f) case 2

step 2   ffi U+FB00: ff U+69: i

feature 'liga', type 'gsub_ligature', lookup 's_s_9', replacing
U+0FB00 (f_f) upto U+00069 (i) by ligature U+0FB03 (f_f_i)
case 2

result   ffi U+FB03: ffi

Features

**font** 78: texgyrepagella-regular.otf @ 48.0pt

features analyze=yes, autolanguage=position, autoscript=position,
checkmarks=yes, curs=yes, devanagari=yes, dummies=yes,
extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes,
liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes,
mode=node, script=dflt, spacekern=yes, tlig=yes, trep=yes

step 1 ffi U+66: f U+66: f U+69: i

feature 'liga', type 'gsub_ligature', lookup 's_s_9', replacing
  U+00066 (f) upto U+00066 (f) by ligature U+0FB00 (f_f) case 2

step 2 ffi U+FB00: ff U+69: i

feature 'liga', type 'gsub_ligature', lookup 's_s_10',
  replacing U+0FB00 (f_f) upto U+00069 (i) by ligature U+0FB03
  (f_f_i) case 2
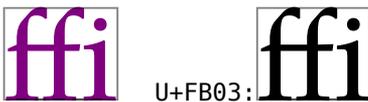
result ffi U+FB03: ffi

5.2.8 Single positioning

Single positioning is also known as kerning, moving characters closer together so that
we get a more uniformly spaced sequence of glyphs. It is a mistake to think that kerning
is always needed! There are fonts that have hardly any kerns or no kerns at all and still
look good.

Dejavu Serif: 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. (E.R. Tufte)

Cambria: We thrive in information–thick worlds because of our marvelous and everyday capacity to se-lect, edit, single out, structure, highlight, group, pair merge, harmonize, synthesize, focus, organize, con-dense, reduce, boil down, choose, categorize, catalog, classify list, abstract, scan, look into, idealize, iso-late, 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. (E.R. Tufte)

Latin Roman Regular: 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. (E.R. Tufte)

Lucida Bright: 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. (E.R. Tufte)

Pagella Regular: 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. (E.R. Tufte)

The next couple of examples show the action for a few words:

font   83: DejaVuSerif.ttf @ 24.0pt

features analyze=yes, autolanguage=position, autoscript=position,
              checkmarks=yes, curs=yes, devanagari=yes, dummies=yes,
              extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes,
              liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes,
              mode=node, script=dflt, spacekern=yes, tlig=yes, trep=yes

step 1   We thrive U+57:W U+65:e [glue] U+74:t U+68:h U+72:r U+69:i U+76:v U+65:e

              feature 'kern', type 'gpos_pair', lookup 'p_s_2', shifting
                single U+00065 (e) by -1.95703pt

result   We thrive U+57:W [kern] U+65:e [glue] U+74:t U+68:h U+72:r U+69:i U+76:v U+65:e

font   45: cambria.ttf @ 24.0pt

features   analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes, mode=node, script=latn, spacekern=yes, tlig=yes, trep=yes

step 1   We thrive   U+57:W U+65:e [glue] U+74:t U+68:h U+72:r U+69:i U+76:v U+65:e

feature 'kern', type 'gpos_pair', lookup 'p_s_0', shifting single U+00065 by -1.40625pt

feature 'kern', type 'gpos_pair', lookup 'p_s_0', shifting single U+00076 by -0.48047pt

feature 'kern', type 'gpos_pair', lookup 'p_s_0', shifting single U+00065 by -0.46875pt

result   We thrive   U+57:W [kern] U+65:e [glue] U+74:t U+68:h U+72:r U+69:i [kern] U+76:v [kern] U+65:e

font   84: lmroman10-regular.otf @ 24.0pt

features   analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes, mode=node, script=dflt, spacekern=yes, tlig=yes, trep=yes

step 1   We thrive   U+57:W U+65:e [glue] U+74:t U+68:h U+72:r U+69:i U+76:v U+65:e

feature 'kern', type 'gpos_pair', lookup 'p_s_1', shifting single U+00065 (e) by -1.992pt

feature 'kern', type 'gpos_pair', lookup 'p_s_1', shifting single U+00065 (e) by -0.672pt

result   We thrive   U+57:W [kern] U+65:e [glue] U+74:t U+68:h U+72:r U+69:i U+76:v [kern] U+65:e

font   85: LucidaBrightOT.otf @ 24.0pt

features   analyze=yes, autolanguage=position, autoscript=position,
checkmarks=yes, curs=yes, devanagari=yes, dummies=yes,
extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes,
liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes,
mode=node, script=dflt, spacekern=yes, tlig=yes, trep=yes

result   We thrive   U+57:W U+65:e [glue] U+74:t U+68:h
U+72:r U+69:i U+76:v U+65:e

font   86: texgyrepagella-regular.otf @ 24.0pt

features   analyze=yes, autolanguage=position, autoscript=position,
checkmarks=yes, curs=yes, devanagari=yes, dummies=yes,
extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes,
liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes,
mode=node, script=dflt, spacekern=yes, tlig=yes, trep=yes

step 1   We thrive   U+57:W U+65:e [glue] U+74:t U+68:h
U+72:r U+69:i U+76:v U+65:e

feature 'kern', type 'gpos_pair', lookup 'p_s_1', shifting
  single U+00065 (e) by -1.56pt

feature 'kern', type 'gpos_pair', lookup 'p_s_1', shifting
  single U+00068 (h) by 0.36pt

feature 'kern', type 'gpos_pair', lookup 'p_s_1', shifting
  single U+00065 (e) by -0.96pt

result   We thrive   U+57:W [kern] U+65:e [glue] U+74:t
[kern] U+68:h U+72:r U+69:i U+76:v [kern] U+65:e

5.2.9 Pairwise positioning

This variant of positioning involved the first, second or both glyphs of a glyph pair. The
specification can influence the horizontal and vertical positions we well as the widths of
the positioned glyphs.

We need an example here.

5.2.10 Mark positioning

Marks are (often) small symbols that represent accents (in latin) or vowels (in arabic)
that get attached to base glyphs. In the input stream they come after the character
that they apply to. Many fonts come with precomposed latin characters which means
that an à in the input is mapped directly onto its corresponding shape. When the input

contains an a followed by a ` input normalization will normally turn this into an à. But, when this doesn't happen, the font machinery has to make sure that the mark gets positioned right onto the base character. In traditional Type1 fonts that more or less happened automatically by overlaying the shapes. In OpenType (single) positioning is used to place the mark right.

```
\showotfcomposition{dejavu-serif*default        at 24pt}{}{a` a\utfchar{"0300} à}
\showotfcomposition{cambria*default             at 24pt}{}{a` a\utfchar{"0300} à}
\showotfcomposition{lmroman10regular*default    at 24pt}{}{a` a\utfchar{"0300} à}
\showotfcomposition{lucidabrightot*default      at 24pt}{}{a` a\utfchar{"0300} à}
\showotfcomposition{texgyrepagellaregular*default at 24pt}{}{a` a\utfchar{"0300} à}
```

Of course a font can contain logic that replaces a sequence of base and mark into pre-composed characters with the right Unicode entry.

font   83: DejaVuSerif.ttf @ 24.0pt

features analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes, mode=node, script=dflt, spacekern=yes, tlig=yes, trep=yes

step 1   à à à U+61: a U+300: [glue] U+61: a U+300: [glue]
U+E0: à

feature 'mark', type 'gpos_mark2base', lookup 'p_s_1', anchor , bound 1, anchoring mark U+00300 (gravecomb) to basechar U+00061 (a) => (1.64063pt,0pt)

feature 'mark', type 'gpos_mark2base', lookup 'p_s_1', anchor , bound 2, anchoring mark U+00300 (gravecomb) to basechar U+00061 (a) => (1.64063pt,0pt)

result   à à à U+61: a U+300: [glue] U+61: a U+300: [glue]
U+E0: à

font   45: cambria.ttf @ 24.0pt

features analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes, mode=node, script=latn, spacekern=yes, tlig=yes, trep=yes

result   à à à U+61: a U+300: [glue] U+61: a U+300: [glue] U+E0: à

font   84: lmroman10-regular.otf @ 24.0pt

features   analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes, mode=node, script=dflt, spacekern=yes, tlig=yes, trep=yes

result   à à à U+61: a U+300: [glue] U+61: a U+300: [glue] U+E0: à

font   85: LucidaBrightOT.otf @ 24.0pt

features   analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes, mode=node, script=dflt, spacekern=yes, tlig=yes, trep=yes

step 1   à à à U+61: a U+300: [glue] U+61: a U+300: [glue] U+E0: à

    feature 'mark', type 'gpos_mark2base', lookup 'p_s_0', anchor , bound 1, anchoring mark U+00300 (gravecomb) to basechar U+00061 (a) => (-0.768pt,0pt)

    feature 'mark', type 'gpos_mark2base', lookup 'p_s_0', anchor , bound 2, anchoring mark U+00300 (gravecomb) to basechar U+00061 (a) => (-0.768pt,0pt)

result   à à à U+61: a U+300: [glue] U+61: a U+300: [glue] U+E0: à

font   86: texgyrepagella-regular.otf @ 24.0pt

features   analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mark=yes, mathkerns=yes, mathrules=yes, mkmk=yes, mode=node, script=dflt, spacekern=yes, tlig=yes, trep=yes

result   à à à U+61: a U+300: [glue] U+61: a U+300: [glue] U+E0: à

You can imagine that when marks are bound to characters that have become ligatures the anchoring is more complex as the font machinery has to keep track of onto which component the mark goes. For this purpose marks as well as base characters and base ligatures have anchors and feature lookups can explicitly refer to them.

5.2.11 Contextual analysis

What actually happens when turning a list of characters into a list of glyphs can range from real simple to pretty complex. For instance the smcp feature only has to run over the list and relate characters to a smallcaps shape. A slightly more complex feature might also demand some positioning. One step further is the use of contextual analysis, i.e. looking at previous, current and following characters (or glyphs). Because features can range from simple to complex the actual processing is not per feature! A font comes with a sequence of so called lookups that relate to a feature, script and language. Also, each feature can use one–to–one, multiple–to–one and many–to–many replacements as well as relative positioning.

So, what actually happens is not that a feature is processed, but that all features are dealt with at the same time, in the order that the font prescribes. Enabling a specific feature means that a step is executed, while a disabled feature skips all steps that are tagged as belonging to that feature. And, as each feature can use contextual analysis, you can imagine that the effective sequence of actions can be a complex mix.

A nice example of a contextual substitution is the centered period character in catalan in ebgaramond:

```
\definefontfeature
  [example]
  [default]
  [locl=yes,script=latn,language=cat]

\definedfont[file:ebgaramond12-regular.otf*default at 40pt]l·l\quad
\definedfont[file:ebgaramond12-regular.otf*example at 40pt]l·l
```

We show the boundingbox of the glyphs. The centered period between two l's is is replaced by a raised variant with no width.

It will be clear that in order to use such features you need to know what the font provides. For some fonts you need to explicitly enable the latin script (while others use default). Such a feature can be part of localized support but that is no rule. In that respect OpenType features are a rather unpredictable mess. For instance, nothing prevents such a feature to be a ligature, and in case you find that strange, especially ligature features are often abused for any purpose.

5.2.12 Ligatures and hyphenation

In this section we will say a few words on how hyphenation interferes with (especially) ligature building. For this you need to know that:

```
effe
```

But when hyphenation is permitted between the two s's we actually have internally:

```
ef{-}{}{}fe
```

The first snippet comes at the end of a line, the second at the beginning of a the next line and the last snippet is used when no hyphenation is needed. Such triplets need to be taken into account when we do replacements and positioning and also when we do contextual lookups.

An OpenType font is just a container that collects the following:

- graphic representations of characters and symbols
- information about what characters the shapes represent
- rules about converting (sequences of) characters into one or more representations
- rules about positioning representations relative to each other

Although the way this information is stored is standardized, the rules are not. You can imagine that there would be some standard way to turn an f and i into an 'fi' but we already saw that this is not the case. Here are some possibilities:

- The two characters get their own standard glyph, maybe with some kerning.
- The two characters are combined into one shape.
- The f gets a narrow representation and is kept close to the standard i.
- A standard f is kerned with a dotless i (not to be confused with the Unicode character).
- A special f is combined with a special i.

If the two characters are represented by their own shape, some contextual analysis takes place. Again there are several approaches to this:

- When an f is seen in the input, the next character is checked and one or both gets replaced.
- When an i is seen in the input, the previous character is checked and the i gets replaced.
- When an f several following characters are checked, for instance to see if we need to take ij into account.

Traditionally the f followed by an f, l and i get a treatment, but some fonts also combine the f with k, j, b, t and more.

The MkIV font handler is rather generic in the sense that it support what the font requires. However, a complication is that the scripts (languages) that use these diverse methods also expect hyphenation within such a ligature. Script like Arabic that are more demanding don't hyphenate so there interference with hyphenation is not a problem.

Some ligatures are sensitive for languages. In languages that have compound words it might be undesirable to have a ligature at a word boundary, or in the Dutch word fijn

we like to have a nice glyph (or combinations) for `ij` but no `fi` ligature. In a similar way hyphenation patterns can have rules and it will be no surprise that the hyphenation mechanism can compete with the ligature building for the best solution. This gets complicated by the fact that there is no real way to recognize in the font handler if we really are dealing with ligature building. Not only is the `liga` feature (and deep down the ligature gsub handling) not bound to ligatures (but simply a many-to-one mapper), some of the mentioned pseudo ligature builders use simple substitution and kerning and there is no way to recognize that as a ligature.

Although it is possible to come up with a solution that is acceptable for many cases, there is no way to predict what kind of tricks font designers will use. A hyphenation point can be seen as follows:

```
effe       ef-fe        e{f-}{f}{ff}e
efficient  ef-fi-cient  e{ffi-}{}{ffi}cient
```

In the second case the larger ligatures has replaced the previous one. We could have kept the first one because there are ways to manage two-step bounding ligatures but it's not worth the trouble (read: way more complex code and increased runtime for the whole mechanism). Here the `{ff}` and `{ffi}` can be individual shapes or just one shape.

The three components of a hyphenation point: the pre, post and replacement text need to be looked at independently so that we get the proper kerning with the preceding and following characters. Also, in more complex (chained) lookups we need to compare each element with its surrounding. A fully expanded solution tree is too time consuming so we take some shortcuts and limits the checks to the level that it has no big impact on performance. The occasionally needed backtracking and inspection of components is currently quite reasonable. We need to trade quality with convenience: the result should look okay but processing speed should also be as high as possible. There is no need to let other scripts or regular fonts suffer too much from excessive script demands of fonts that could have be done better.

The complication is that we not only need to check and replace but also need to check the kerning with preceding and following characters. We also need to take the hyphen into account (here one, but there can also be one after the break.

It is for this reason that in MkIV we have a (we think) acceptable mix of heuristics around hyphenation points that deal with single and multiple substitution as well as kerning. It will never be 100% pertect but we consider it better to drop an occasional hyphenation in favor of proper font handling. In practice TEX is clever enough to break a paragraph in lines within these restrictions.

In ConTEXt we have the traditional TEX hyphenator but also provide an extensible Lua reimplementation. That one might become the default in future versions. In traditional TEX there are several low level hyphenation representations: simple hyphen only points, injected by the hyphenator, explicitly injected by the user or originating from a hyphen

character. Then there is the generic (pre, post, replace) discretionary that can be explicitly injected by the user (or a macro). In MkIV all hyphenation points get normalized to this generic discretionary. There is no need for old-time optimizations and a consistent (expanded) representation is easier to deal with in other extensions. However, because the font handler is supposed to also work outside ConTEXt we need to deal with traditional cases too. But ... the results might differ a bit.

5.2.13 Color

A recent new (and evolving) addition to OpenType is colored glyphs. One variant (by Microsoft) uses overlays and this method is quite efficient.

```
\definefontfeature[colored][colr=yes]
\definefontsynonym[Emoji][file:seguiemj.ttf*default,colored]

\definesymbol[bug][\getglyphdirect{Emoji}{\char"1F41B}]
\definesymbol[ant][\getglyphdirect{Emoji}{\char"1F41C}]
\definesymbol[bee][\getglyphdirect{Emoji}{\char"1F41D}]
```

Here we see a 🐛, 🐜 and 🐝, and they come in color! Once Unicode started adding such symbols (and more get added) the distinction between characters and symbols get even fuzzier. Of course one can argue that we communicate in pictograms but even then, given that mankind lasts a while, the Unicode repertoire will explode.

U+1F41B: bug   U+1F41C: ant   U+1F41D: bee
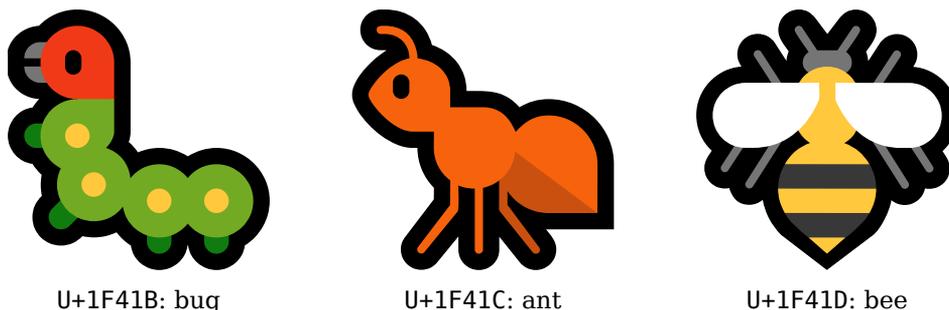
Figure 5.1   A few emojis from `seguiemj.ttf`

Here we use `seguiemj.ttf`, a font that comes with MS Windows. Colors are achieved by combining glyphs rendered in different colors. A variant that uses svg instead of overlays is `emojionecolor-svginot.ttf`:

```
\definefontfeature[svg][svg=yes]
\definefontsynonym[Emoji][file:emojionecolor-svginot.ttf*default,svg]
```

This time we get 🐛, 🐜 and 🐝 and they look quite different. Both fonts also have ligatures and you can wonder what sense that makes. It makes it impossible to swap fonts and as there is no standard one never knows what to expect.

U+1F41B: bug   U+1F41C: ant   U+1F41D: bee
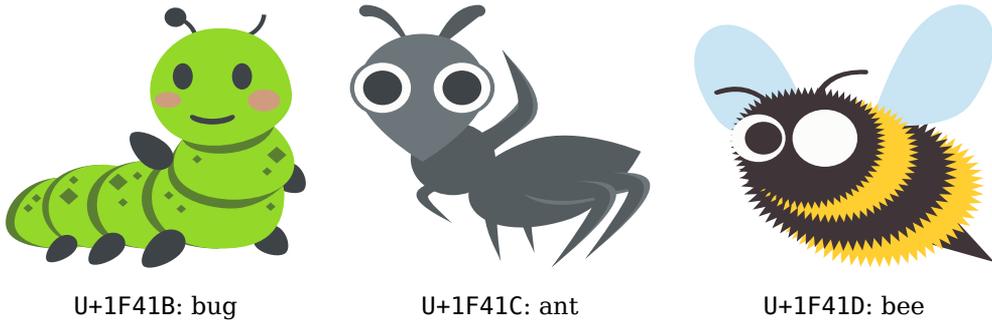
Figure 5.2   A few emojis from `emojionecolor-svginot.ttf`

How do we know what faces add up to the ligature 👨‍👩‍👧‍👦 and how are we supposed to know that there should `zwj` in between? When we input four faces separated by zero width joiners, we get a four face symbol instead. The reason for having the joiners in between is probably to avoid unexpected ligatures. The sequence `man`, `woman`, `boy`, `boy` gives `family`: 👨 + `zwj` 👩 + `zwj` 👦 + `zwj` 👦 = 👨‍👩‍👦‍👦, but two girls also work: 👨 + `zwj` 👩 + `zwj` 👧 + `zwj` 👧 = 👨‍👩‍👧‍👧, so does a mixture of kids: 👨 + `zwj` 👩 + `zwj` 👧 + `zwj` 👦 = 👨‍👩‍👧‍👦, although (at least currently): 👨 + `zwj` 👩 + `zwj` 👦 + `zwj` 👧 = 👨👩👦👧, gives twin boys. Of course the real family emoj is 👪.

In our times for sure many combinations are possible, so: 👨 + `zwj` 👨 + `zwj` 👧 + `zwj` 👧 = 👨‍👨‍👧‍👧, indeed gives a family, but I wonder at what point cultural bias will creep into font design. One can even wonder how clothing and haircut will demand frequent font updates: 👬, 👭, 👫.

In the math alphabets we have a couple of annoying holes because some characters were already present in Unicode. The bad thing here is that we now always have to deal with these exceptions. But not so with emojis because here eventually all variants will show up. Where a character A in red or blue uses the same code point, a white telephone and black telephone have their own. And because obsolete scripts are already supported in Unicode and more get added, we can expect old artifacts also showing up at some time. Soon the joystick 🕹 will be an unknown item to most of us, while the Microsoft hololens migth get its slot.

For sure these mechanisms will evolve and to what extent we support them depends on what users want. At least we have the basics implemented.

5.3 Extras

Todo.

5.4 Goodies

Goodies range from simple to complex. They share that they are defined in files and loaded at runtime. There is a good change that when you read this, that there are already

U+1F423: hatching chick   U+1F424: baby chick   U+1F425: front-facing
                                                     baby chick

Figure 5.3   Will all animals come in stages of development?

more goodies than mentioned here. Here we will just mention a couple of goodies. More details can be found in the files that ship with ConTeXt and have suffix `lfg`.

A goodie file is a regular Lua file and is supposed to return a table. This table collects data that is used for implementing the goodie or relates to a regular feature. It can also provide information that is used for patching a font. An example of a simple goodie file is the ones that accompanies the first release of the OpenType Lucida fonts.

```
return {
  name = "lucida-opentype-math",
  version = "1.00",
  comment = "Goodies that complement lucida opentype.",
  author = "Hans Hagen",
  copyright = "ConTeXt development team",
  mathematics = {
    alternates = {
      italic = {
        feature = 'ss01',
        value = 1,
        comment = "Mathematical Alternative Italic"
      },
    }
  }
}
```

This goodie file is only providing information about the meaning of a stylistic alternate. These have abstract tags like `ss01` and in this case this category collects alternative italic (calligraphic) shapes. Because math does not follow the same rules as text, this feature is enabled explicitly.

In the goodie file of Xits math the alternates table has more entries:

```
alternates = {
  cal       = { ... comment = "Mathematical Calligraphic Alphabet" },
```

```
  greekssup = { ... comment = "Mathematical Greek Sans Serif Alphabet" },
  greekssit = { ... comment = "Mathematical Italic Sans Serif Digits" },
  monobfnum = { ... comment = "Mathematical Bold Monospace Digits" },
  mathbbbf  = { ... comment = "Mathematical Bold Double-Struck Alphabet" },
  mathbbit  = { ... comment = "Mathematical Italic Double-Struck Alphabet" },
  mathbbbi  = { ... comment = "Mathematical Bold Italic Double-Struck Alphabet" },
  upint     = { ... comment = "Upright Integrals" },
  vertnot   = { ... comment = "Negated Symbols With Vertical Stroke" },
}
```

An alternate is triggered at the TEX end with:

```
$ABC$ $\cal ABC$ $\mathalternate{cal}\cal ABC$
```

This is an example of a dynamic feature that gets applied when enabled at a specific location in the input. The `cal` is only recognized when it is defined in a goodies file, where the value is defined (in all of the above cases the value is 1).

The Xits math fonts has a goodie files that starts with:

```
return {
  name = "xits-math",
  version = "1.00",
  comment = "Goodies that complement xits (by Khaled Hosny).",
  author = "Hans Hagen",
  copyright = "ConTeXt development team",
  mathematics = {
    italics = {
      ["xits-math"] = italics,
    },
    alternates = {
```

Here the `italics` variable is a table defined before the `return` that looks as follows:

```
local italics = {
  defaultfactor = 0.025,
  disableengine = true,
  corrections   = {
   -- [0x1D44E] = 0.99,    -- a (fraction of quad)
   -- [0x1D44F] = 100,     -- b (font points)
      [0x1D453] = -0.0375, -- f
  }
}
```

This rather specific table tells ConTEXt that (when enabled) it has to apply italic correction. It disables support built into the TEX engine (which in the case of LuaTEX is close

to absent anyway). It will apply a default italic correction of `0.025` but for some shapes a different value is used. Again we have some commands at the T_EX end:

```
\setupmathematics[italics=1] % fontitalics
\setupmathematics[italics=2] % fontdata
\setupmathematics[italics=3] % quad based
\setupmathematics[italics=4] % combination of 1 and 3
```

Because (definitely at the start of the LuaT_EX project) we had no proper OpenType math fonts, but at the same time wanted to move on to OpenType and Unicode math and no longer struggle with all those math families and definitions. The way out of this problem is to define a virtual math font. The code for doing this is built into the MkIV core but is controlled by a goodie definition. Take for instance Antykwa Math:

```
return {
  name = "antykwa-math",
  version = "1.00",
  comment = "Goodies that complement antykwa math.",
  author = "Hans, Mojca, Aditya",
  copyright = "ConTeXt development team",
  mathematics = {
    mapfiles = {
      "antt-rm.map",
      "antt-mi.map",
      "antt-sy.map",
      "antt-ex.map",
      "mkiv-base.map",
    },
    virtuals = {
      ["antykwa-math"] = {
        { name = "file:AntykwaTorunska-Regular", features = "virtualmath", main = true },
        { name = "mi-anttri.tfm", vector = "tex-mi", skewchar=0x7F },
        { name = "mi-anttri.tfm", vector = "tex-it", skewchar=0x7F },
        { name = "sy-anttrz.tfm", vector = "tex-sy", skewchar=0x30, parameters = true } ,
        { name = "ex-anttr.tfm", vector = "tex-ex", extension = true } ,
        { name = "msam10.tfm", vector = "tex-ma" },
        { name = "msbm10.tfm", vector = "tex-mb" },
      },
```

Normally users will not define such tables but the keys give an indication of what is involved. The same is true for the previously shown tables: they are just provided in the ConT_EXt distribution.

Text fonts also can have goodies. We start with a rather dumb one and there will be not that many of those. This one is needed to turn a Type1 font with a rather special encoding into a Unicode font. The next mapping is possible because the dingbats are part of Unicode.

```
return {
  name = "dingbats",
  version = "1.00",
  comment = "Goodies that complement dingbats (funny names).",
  author = "Hans Hagen",
  copyright = "ConTeXt development team",
  remapping = {
    tounicode = true,
    unicodes = {
      a1   = 0x2701,
      a10  = 0x2721,
      a100 = 0x275E,
      a101 = 0x2761,
      a102 = 0x2762,
```

Applying this encoding happens in two steps. Because goodies like this are just features, we need to define a proper font feature set:

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

We have a base mode font, so no special processing takes place. The goodies key is used to communicate the goodies file. The unicoding key is used to apply the encoding. Of course this only works because the remapper code is present in the core and is hooked in to the font initialization code. The dingbats feature set is predefined, just as the font definition:

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

Here is a goodie file that I made a while ago:

```
return {
  name = "oxoniensis",
  version = "1.00",
  comment = "Oxoniensis test file for Thomas Schmitz.",
  author = "Hans Hagen",
  copyright = "ConTeXt development team",
  features = {
    lunatesigma = {
      type = "substitution",
      data = {
        sigma  = 0x03F2,
        sigma1 = 0x03F2,
```

```
        Sigma  = 0x03F9,
        phi    = phi1,
      },
    }
  },
}
```

There is not that much to say about this, apart from that it's a sort of fake feature that gets enabled as regular one:

```
\definefontfeature[test]
  [mode=node,
   kern=yes,
   lunatesigma=yes,
   goodies=oxoniensis]

\definefont[somefont][file:oxoniensis*test]
```

A complete different kind of goodie is the following. At one of the ConTEXt meetings Mojca Miklavec discussed the possibility to have an additional mechanism for defining combinations of fonts. Often fonts come in a set of four (regular, italic, bold and bold italic). In MkII the complexity of typescripts depends on the amount of encodings that need to be supported but in MkIV things are easier. For a set of four fonts a typescript looks as follows:

```
\starttypescript [sans] [somesansfont] [name]
  \setups[font:fallback:sans]
  \definefontsynonym [Sans]           [file:somesans]  [features=default]
  \definefontsynonym [SansBold]       [file:somesansb] [features=default]
  \definefontsynonym [SansItalic]     [file:somesansi] [features=default]
  \definefontsynonym [SansBoldItalic] [file:somesansz] [features=default]
\stoptypescript
```

We still have the abstract notion of a Sans font so that we can refer to the regular shape without knowing the real name but the number of lines needed is small. Such a definition can then be referred to using:

```
\starttypescript[somefontset]
  \definetypeface [somefontset] [rm] [serif] [someserif] [default]
  \definetypeface [somefontset] [ss] [sans]  [somesans]  [default]
  \definetypeface [somefontset] [tt] [mono]  [somemono]  [default]
  \definetypeface [somefontset] [mm] [math]  [somemath]  [default]
\stoptypescript
```

So far things look simple. Given that many fonts follow a similar naming scheme Wolfgang made a module that avoids such definitions altogether. However, being involved in the development of the Antykwa fonts, Mojca ran into the situation that not just four

fonts were part of the set but many more. There are several weight (think of light and heavy variants) as well as condensed variants and of course the whole set is not per se a multiple of four.

In the meantime, in addition to the `file:` and `name:` accessors, ConTEXt had an additional one tagged `spec:` where a string made out of weight, style, width etc. is turned into a (best guessed) font name. Therefore the most natural way to deal with the many-fonts-in-a-set dilemma was to provide an additional interface between this specification and the font set and the most robust method was to define all in a goodie file.

In this case the goodies are loaded independent of the font, that is: not as a feature. For instance:

```
\loadfontgoodies[antykwapoltawskiego]
```

This file maps obscure fontnames onto the `spec:` interface so that we can access them in a robust way.

```
\definefont
  [MyFontA]
  [file:Iwona-Regular*smallcaps]
\definefont
  [MyFontB]
  [file:AntykwaTorunska-Regular*smallcaps]
\definefont
  [MyFontC]
  [file:antpoltltcond-regular*smallcaps]
\definefont
  [MyFontD]
  [spec:antykwapoltawskiego-bold-italic-condensed-normal*smallcaps]
\definefont
  [MyFontE]
  [spec:antykwapoltawskiego-bold-italic-normal]
```

The goodies file looks as follows:

```
return {
  name = "antykwa-poltawskiego",
  version = "1.00",
  comment = "Goodies that complement Antykwa Poltawskiego",
  author = "Hans & Mojca",
  copyright = "ConTeXt development team",
  files = {
    name = "antykwapoltawskiego", -- shared
    list = {
      ["AntPoltLtCond-Regular.otf"] = {
        weight = "light",
```

```
        style  = "regular",
        width  = "condensed",
      },
      ...
      ["AntPoltExpd-BoldItalic.otf"] = {
        weight = "bold",
        style  = "italic",
        width  = "expanded",
      },
    },
  },
  typefaces = {
    ["antykwapoltawskiego-light"] = {
      shortcut     = "rm",
      shape        = "serif",
      fontname     = "antykwapoltawskiego",
      normalweight = "light",
      boldweight   = "medium",
      width        = "normal",
      size         = "default",
      features     = "default",
    },
    ...
  },
}
```

In addition to the files-to-specification mapping, there is also a typeface specification table. This permits the definition of a typeface in the following way:

```
\definetypeface
  [name=mojcasfavourite,
   preset=antykwapoltawskiego,
   normalweight=light,
   boldweight=bold,
   width=expanded]

\setupbodyfont
  [mojcasfavourite]
```

When resolving the definition, the best possible match will be taken from the typeface table in the goodie file. Of course this is not something that we expect the average user to deliver and deal with.

As the Antykwa font is somewhat atypical and not used in everyday typesetting, you might wonder if all this overhead makes sense. However, there are type foundries that

do ship their fonts in many weights and for those using a Lua goodie file instead of many typescripts in TEX coding makes sense. Take for instance TheMix:

```
\loadfontgoodies
  [themix]

\definetypeface
  [name=themix,
   preset=themix-light]

\definetypeface
  [name=themix,
   preset=themixmono-light]

\setupbodyfont
  [themix]
```

In this case the goodie file can serve as a template for more such fonts. In order to be efficient this goodie file uses a couple of local tables (we could have used metatables instead).

```
local themix = {
  name    = "themix",
  shortcut = "ss",
  shape   = "sans",
  fontname = "themix",
  width   = "normal",
  size    = "default",
  features = "default",
}

local themixmono = {
  name    = "themixmono",
  shortcut = "tt",
  shape   = "mono",
  fontname = "themixmono",
  width   = "normal",
  size    = "default",
  features = "default",
}
```

The main goodie table defines a lot of weights:

```
return {
  name = "themix",
  version = "1.00",
```

```
          comment = "Goodies that complement TheMix (by and for sale at www.lucasfonts.com).",
          author = "Hans Hagen",
          copyright = "ConTeXt development team",
          files = {
            list = {
              ["TheMixOsF-ExtraLight"] = {
                name   = "themix",
                weight = "extralight",
                style  = "regular",
                width  = "normal"
              },
              ["TheMixOsF-ExtraLightItalic"] = {
                ...
              },
              ...
              ["TheMixOsF-Black"] = {
                ...
              },
              ["TheMixOsF-BlackItalic"] = {
                ...
              },
              ...
              --
              ["TheMixMono-W2ExtraLight"] = {
                name   = "themixmono",
                weight = "extralight",
                style  = "regular",
                width  = "normal"
              },
              ...
              ["TheMixMono-W9BlackItalic"] = {
                ...
              },
            },
          },
          typefaces = {
            ["themix-extralight"] = table.merged(themix, {
              normalweight = "extralight",
              boldweight   = "semilight"
            }),
            ["themix-light"] = table.merged(themix, {
              normalweight = "light",
              boldweight   = "normal"
            }),
```

```
    ...
    ["themixmono-bold"] = table.merged(themixmono, {
      normalweight = "bold",
      boldweight   = "black"
    }),
  },
}
```

It's now time for some generic goodies. In the ConTeXt distribution there is a goodie file that (at the time of this writing) looks as follows:

```
local default = {
  analyze  = "yes",
  mode     = "node",
  language = "dflt",
  script   = "dflt",
}

local smallcaps = {
  smcp = "yes",
}

local function statistics(tfmdata)
  commands.showfontparameters(tfmdata)
end

local function squeeze(tfmdata)
  for k, v in next, tfmdata.characters do
    v.height = 0.75 * (v.height or 0)
    v.depth  = 0.75 * (v.depth  or 0)
  end
end

return {
  name = "demo",
  version = "1.01",
  comment = "An example of goodies.",
  author = "Hans Hagen",
  featuresets = {
    default = {
      default,
    },
    smallcaps = {
      default, smallcaps,
    },
```

```
    },
    colorschemes = {
      default = {
        [1] = {
          "one", "three", "five", "seven", "nine",
        },
        [2] = {
          "two", "four", "six", "eight", "ten",
        },
      },
      all = {
        [1] = {
          "*",
        },
      },
      some = {
        [1] = {
          "0x0030:0x0035",
        },
      },
    },
    postprocessors = {
      statistics = statistics,
      squeeze    = squeeze,
    },
}
```

This demo file implements several goodies: featuresets, colors and postprocessors. Keep in mind that a goodie file can provide whatever information it wants but of course only known subtables will be dealt with.

The coloring of glyphs can happen by name, which assumes that glyph names are used, or by number. Here we use generic glyph names, but for a specific font one might need to provide a special goodie file. For instance, the color section of the goodie file for the husayni font has entries like:

```
[3] = {
  "Ttaa.waqf", "SsLY.waqf", "QLY.waqf", "Miim.waqf", "LA.waqf", "Jiim.waqf",
  "Threedotsabove.waqf", "Siin.waqf", "Ssaad.waqf", "Qaaf.waqf", "SsL.waqf",
  "QF.waqf", "SKTH.waqf", "WQFH.waqf", "Kaaf.waqf", "Ayn.ruku", "Miim.nuun_high",
  "Siin.Ssaad", "Nuunsmall", "emptydot_low", "emptydot_high", "Sifr.fill",
  "Miim.nuun_low", "Nuun.tanwiin",
},
```

Of course such a definition can only be made when the internals of the font are known and in this case it concerns a pretty complex font.

Features

```
\definefontfeature
  [demo-colored]
  [goodies=demo,
   colorscheme=default,
   featureset=default]

\definefontfeature
  [demo-colored-all]
  [goodies=demo,
   colorscheme=all,
   featureset=default]

\definefontfeature
  [demo-colored-some]
  [goodies=demo,
   colorscheme=some,
   featureset=default]

\definefont[DemoFontA][MonoBold*demo-colored      at 10pt]
\definefont[DemoFontB][MonoBold*demo-colored-all  at 10pt]
\definefont[DemoFontC][MonoBold*demo-colored-some at 10pt]

\starttabulate[|||||]
\NC
    \DemoFontA \resetfontcolorscheme   test 1234567890 \NC
    \DemoFontA \setfontcolorscheme  [1]test 1234567890 \NC
    \DemoFontA \setfontcolorscheme  [2]test 1234567890 \NC
\NR
\NC
    \DemoFontB \resetfontcolorscheme   test 1234567890 \NC
    \DemoFontB \setfontcolorscheme  [1]test 1234567890 \NC
    \DemoFontB \setfontcolorscheme  [2]test 1234567890 \NC
\NR
\NC
    \DemoFontC \resetfontcolorscheme   test 1234567890 \NC
    \DemoFontC \setfontcolorscheme  [1]test 1234567890 \NC
    \DemoFontC \setfontcolorscheme  [2]test 1234567890 \NC
\NR
\stoptabulate
```

test 1234567890 test 1234567890 test 1234567890
test 1234567890 test 1234567890 test 1234567890
test 1234567890 test 1234567890 test 1234567890

Here is an example that I made at the TUG 2013 conference in Japan, after a presentation by Chof. The font (adapted by by Dohyun Kim) can be downloaded from: http://ftp.ktug .org/KTUG/hcr-lvt/1.910_nomac/.

```
\definefontfeature
  [korean-composed]
  [goodies=hanbatanglvt,
   colorscheme=default,
   mode=node,
   ljmo=yes,
   tjmo=yes,
   vjmo=yes,
   script=hang,
   language=kor]

\definefont
  [KoreanJMO]
  [hanbatanglvt*korean-composed]

\definecolor[colorscheme:100:1][r=.75]
\definecolor[colorscheme:100:2][g=.75]
\definecolor[colorscheme:100:3][b=.75]

\definecolor[colorscheme:101:1][g=.75,b=.75]
\definecolor[colorscheme:101:2][r=.75,b=.75]
\definecolor[colorscheme:101:3][r=.75,g=.75]
```

나랏말쓰미中듕國귁에달아문쫑와로서르ᄉ못디아니ᄒᆞᆯ씨사ᄅᆞᆷ마다히ᅇᅧ수비니겨나…

no colorscheme

나랏말쓰미中듕國귁에달아문쫑와로서르ᄉ못디아니ᄒᆞᆯ씨사ᄅᆞᆷ마다히ᅇᅧ수비니겨나…

colorscheme 100

나랏말쓰미中듕國귁에달아문쫑와로서르ᄉ못디아니ᄒᆞᆯ씨사ᄅᆞᆷ마다히ᅇᅧ수비니겨나…

colorscheme 101

The goodie definition looks as follows (watch how we use ranges):

```
return {
    name = "hanbatanglvt",
    version = "1.00",
    comment = "Goodies that complement the hanbatanglvt fonts.",
    author = "Hans Hagen",
    colorschemes = {
```

```
    default = {
        { "0x01100:0x0115F" }, -- jamo_initial (r/c)
        { "0x01160:0x011A7" }, -- jamo_medial  (g/m)
        { "0x011A8:0x011FF" }, -- jamo_final   (b/y)
    }
  }
}
```

This is much shorter (and efficent) that defining a whole vector, as in:

```
local f_uni_base = string.formatters["uni%04X"]
local f_uni_plus = string.formatters["uni%04X.y%s"]

local function range(first,last)
    local t = { }
    for i=first,last do
        t[#t+1] = f_uni_base(i)
        for j=0,19 do
            t[#t+1] = f_uni_plus(i,j)
        end
    end
    return t
end

return {
    name = "hanbatanglvt",
    version = "1.00",
    comment = "Goodies that complement the hanbatanglvt fonts.",
    author = "Hans Hagen",
    colorschemes = {
        default = {
            range(0x01100,0x0115F), -- jamo_initial (r/c)
            range(0x01160,0x011A7), -- jamo_medial  (g/m)
            range(0x011A8,0x011FF), -- jamo_final   (b/y)
        }
    }
}
```

By using names we don't depend on Unicode which sometimes is needed when glyphs have ended up in the private space. However, by default, after glyphs have been mapped to colors, an extra pass will make sure that characters pushed into private space will get the same mapping as their regular Unicode has gotten (given that the number is known). Of course explicitly assigned colors will be preserved.

Another generic demo feature is postprocessing. In principle one can add additional postprocessors but for that the source code needs to be consulted which in turn assumes some knowledge of fonts and ConTEXt internals.

```
\definefontfeature
  [justademoa]
  [default]
  [goodies=demo,
   postprocessor=squeeze]

\definefontfeature
  [justademob]
  [default]
  [goodies=demo,
   postprocessor=statistics]

\definefontfeature
  [justademoc]
  [default]
  [goodies=demo,
   postprocessor={statistics,squeeze}]
```

The statistics just print some font parameters to the log so that one is not showing up here. The squeeze looks as follows:

```
\definefont[DemoFontD][Serif*default   at 30pt]
\definefont[DemoFontE][Serif*justademoa at 30pt]
```

[DemoFontD height & depth](#)   [DemoFontE height & depth](#)

The squeezer just makes the height and depth of glyphs a bit smaller and it is not that hard to imagine other manipulations. The demo goodie file is good place to start playing with such things.

Because there is less standardization with respect to features than one might suspect, goodie files provide a mean to define featuresets. We can use such a set in another definition:

```
\definefontfeature
  [demo-smallcaps]
  [goodies=demo,
   featureset=smallcaps]
```

Of course this only makes sense for more complex combinations. The already mentioned husayni font comes with many features and most of these work together.

The basic goodie table looks as follows:

```
return {
  name      = "husayni",
  version   = "1.00",
```

```
  comment     = "Goodies that complement the Husayni font by Idris Samawi Hamid.",
  author      = "Idris Samawi Hamid and Hans Hagen",
  featuresets = { },
  solutions   = { },
  stylistics  = { },
  colorschemes = { },
}
```

We already saw the color schemes and now we will fill in the other tables. First we define a couple of sets:

```
local basics = {
  analyze  = "yes",
  mode     = "node",
  language = "dflt",
  script   = "arab",
}

local analysis = {
  ccmp = "yes",
  init = "yes", medi = "yes", fina = "yes",
}

local regular = {
  rlig = "yes", calt = "yes", salt = "yes", anum = "yes",
  ss01 = "yes", ss03 = "yes", ss07 = "yes", ss10 = "yes", ss12 = "yes",
  ss15 = "yes", ss16 = "yes", ss19 = "yes", ss24 = "yes", ss25 = "yes",
  ss26 = "yes", ss27 = "yes", ss31 = "yes", ss34 = "yes", ss35 = "yes",
  ss36 = "yes", ss37 = "yes", ss38 = "yes", ss41 = "yes", ss42 = "yes",
  ss43 = "yes", js16 = "yes",
}

local positioning = {
  kern = "yes", curs = "yes", mark = "yes", mkmk = "yes",
}

local minimal_stretching = {
  js11 = "yes", js03 = "yes",
}

local medium_stretching = {
  js12="yes", js05="yes",
}
local maximal_stretching= {
  js13 = "yes", js05 = "yes", js09 = "yes",
```

```
}

local wide_all = {
  js11 = "yes", js12 = "yes", js13 = "yes", js05 = "yes", js09 = "yes",
}

local shrink = {
  flts = "yes", js17 = "yes", ss05 = "yes", ss11 = "yes", ss06 = "yes",
  ss09 = "yes",
}

local default = {
  basics, analysis, regular, positioning, -- xxxx = "yes", yyyy = 2,
}
```

Next we define some featuresets and we use the default as starting point:

```
featuresets = {
  default = {
    default,
  },
  minimal_stretching = {
    default, js11 = "yes", js03 = "yes",
  },
  medium_stretching = {
    default, js12="yes", js05="yes",
  },
  maximal_stretching= {
    default, js13 = "yes", js05 = "yes", js09 = "yes",
  },
  wide_all = {
    default, js11 = "yes", js12 = "yes", js13 = "yes", js05 = "yes",
    js09 = "yes",
  },
  shrink = {
    default, flts = "yes", js17 = "yes", ss05 = "yes", ss11 = "yes",
    ss06 = "yes", ss09 = "yes",
  },
}
```

When defining the font at the TEX end we can now refer to for instance `wide_all` which saves us some typing. However, it does not stop here. In a later paragraph we will see how fonts can work in tandem with the parbuilder. For that purpose the goodie table has a `solutions` subtable:

```
solutions = {
```

```
  experimental = {
    less = {
      "shrink"
    },
    more = {
      "minimal_stretching", "medium_stretching", "maximal_stretching", "wide_all"
    },
  },
}
```

Here we define an experimental solution for optimizing the lines in a paragraph: we can narrow words or we can widen them according to a specific featureset. In order to reach the optimal solution the text will be retypeset under a different feature regime.

TODO: show how to apply.

Because there are a some 55 stylistic and 21 justification variants the goodie file also provides a `stylistics` table and for tracing purposes the colorschemes table is populated.

Yet another demonstration of manipulation is the following. Not all fonts come with all combined glyphs. Although we have an auto-compose feature in ConTeXt it sometimes helps to be specific with respect to some combinations. This is where the `compositions` goodie kicks in:

```
local compose = {
  [0x1E02] = {
    anchored = "top",
  },
  [0x1E04] = {
    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,
      },
    },
  },
}

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,
  }
}
```

Of course this assumes some knowledge of the font metrics (in base points) and Unicode slots, but it might be worth the trouble. After all, one only needs to figure it out once. But keep in mind that it will always be a kludge.

A slightly different way to define such compositions is the following:

```
local defaultunits = 193 - 30

local compose = {
                  DY = defaultunits,
 -- [0x010C] = { DY = defaultunits }, -- Ccaron
 -- [0x02C7] = { DY = defaultunits }, -- textcaron
}

-- fractions relative to delta(X_height - x_height)

local defaultfraction = 0.85

local compose = {
```

```
    DY = defaultfraction, -- uppercase compensation
}

return {
  name = "lucida-one",
  version = "1.00",
  comment = "Goodies that complement lucida.",
  author = "Hans and Mojca",
  copyright = "ConTeXt development team",
  compositions = {
    ["lbr"]  = compose,
    ["lbi"]  = compose,
    ["lbd"]  = compose,
    ["lbdi"] = compose,
  }
}
```

Of course no one really needs this because OpenType Lucida fonts have replaced the Type1 versions.

The next goodie table is dedicated to the de facto standard T_EX font Latin Modern. There is a bit of history behind this file. When we started writing ConT_EXt there were not that many fonts available and so we ended up with a font system that was rather well suited for the predecessor of Latin Modern, called Computer Modern. And because these fonts came in design sizes the font system was made such that it could cope efficiently with many files in a font set. Although there is no additional overhead compared to small font sets, apart from more files, there is some burden in defining them. And, as they are the default fonts, these definitions slow down the initialization of ConT_EXt (which is due to the fact that the large typescript definitions were loaded and parsed). So, at some point the decision was made to kick out these definitions and move the burden of figuring out the right size to Lua. When Latin Modern is chosen as font the effect is the same when design sizes are enabled. But, instead of many definitions (one for each combination of size and style) we now have an option. A non-designsize typeface is defined as follows:

```
\starttypescript [modern,modern-base]
  \definetypeface [\typescriptone] [rm] [serif] [modern] [default]
  \definetypeface [\typescriptone] [ss] [sans]  [modern] [default]
  \definetypeface [\typescriptone] [tt] [mono]  [modern] [default]
  \definetypeface [\typescriptone] [mm] [math]  [modern] [default]
  \quittypescriptscanning
\stoptypescript
```

The designsize variant looks like this:

```
\starttypescript [modern-designsize]
  \definetypeface [\typescriptone]
```

```
    [rm] [serif] [latin-modern-designsize] [default] [designsize=auto]
  \definetypeface [\typescriptone]
    [ss] [sans]  [latin-modern-designsize] [default] [designsize=auto]
  \definetypeface [\typescriptone]
    [tt] [mono]  [latin-modern-designsize] [default] [designsize=auto]
  \definetypeface [\typescriptone]
    [mm] [math]  [latin-modern-designsize] [default] [designsize=auto]
  \quittypescriptscanning
\stoptypescript
```

Of course there are accompanying typescripts that map the sans, serif, mono and math
styles onto files. The designsize magic uses the following table. We show only part of
the file, as it is in the ConTeXt distribution.

```
return {
  name = "latin modern",
  version = "1.00",
  comment = "Goodies that complement latin modern.",
  author = "Hans Hagen",
  copyright = "ConTeXt development team",
  mathematics = {
    tweaks = {
      aftercopying = {
        mathematics.tweaks.fixbadprime, -- prime is too low
      },
    },
  },
  designsizes = {
    ["LMMathRoman-Regular"] = {
      ["4pt"]  = "LMMath5-Regular@lmroman5-math",
      ...
      ["12pt"] = "LMMath12-Regular@lmroman12-math",
      default  = "LMMath10-Regular@lmroman10-math"
    },
    ["LMMathRoman-Bold"] = { -- not yet ready
      ...
    },
    ["LMRoman-Regular"] = {
      ["4pt"]  = "file:lmroman5-regular",
      ...
      ["12pt"] = "file:lmroman12-regular",
      default  = "file:lmroman10-regular",
    },
    ["LMRoman-Bold"] = {
      ...
```

```
    },
    ["LMRoman-Demi"] = {
      default  = "file:lmromandemi10-regular",
    },
    ["LMRoman-Italic"] = {
      ...
    },
    ...
    ["LMRoman-Unslanted"] = {
        default  = "file:lmromanunsl10-regular",
    },
    ["LMSans-Regular"] = {
        ...
    },
    ["LMTypewriter-Regular"] = {
        ...
    },
    ...
    ["LMTypewriterVarWd-DarkOblique"] = {
      default  = "file:lmmonoproplt10-boldoblique",
    },
    ...
    ["LMTypewriter-CapsOblique"] = {
      default  = "file:lmmonocaps10-oblique",
    },
  }
}
```

The auto option will choose a best fit compatible to the MkII implementation. When default is used instead, the default filename will be taken. Of course one might wonder if there will ever be similar goodie files because design sizes are not that popular nowadays.

Not all fonts are perfect and of course the LuaTEX engine can have flaws as well. For this reason we can implement patches. Here is another example of a goodie file that has some more code than just a table:

```
local patches = fonts.handlers.otf.enhancers.patches

local function patch(data,filename,threshold)
  local m = data.metadata.math
  if m then
    local d = m.DisplayOperatorMinHeight or 0
    if d < threshold then
      patches.report("DisplayOperatorMinHeight(%s -> %s)",d,threshold)
      m.DisplayOperatorMinHeight = threshold
```

```
      end
   end
end

patches.register("after","analyze math","asana",
    function(data,filename) patch(data,filename,1350) end)

local function less(value,target,original)
  return 0.25 * value
end

local function more(value,target,original)
  local o = original.mathparameters.DisplayOperatorMinHeight
  if o < 2800 then
      return 2800 * target.parameters.factor
  else
      return value -- already scaled
  end
end

return {
  name = "asana-math",
  version = "1.00",
  comment = "Goodies that complement asana.",
  author = "Hans Hagen",
  copyright = "ConTeXt development team",
  mathematics = {
    parameters = {
      DisplayOperatorMinHeight        = more,
      StackBottomDisplayStyleShiftDown = less,
      StackBottomShiftDown            = less,
      StackDisplayStyleGapMin         = less,
      StackGapMin                     = less,
      StackTopDisplayStyleShiftUp     = less,
      StackTopShiftUp                 = less,
      StretchStackBottomShiftDown     = less,
      StretchStackGapAboveMin         = less,
      StretchStackGapBelowMin         = less,
      StretchStackTopShiftUp          = less,
    }
  }
}
```

In fact, in addition to already mentioned mapfiles and virtuals subtables, we can pass variables and overload parameters.

```
return {
  name = "lm-math",
  ...
  mathematics = {
    mapfiles = {
      ...
    },
    virtuals = {
      ...
    variables = {
      joinrelfactor = 3, -- default anyway
    },
    parameters = { -- test values
     -- FactorA = 123.456,
     -- FactorB = false,
     -- FactorC = function(value,target,original)
     --    return 7.89 * target.factor
     -- end,
     -- FactorD = "Hi There!",
    },
  }
}
```

This kind of goodie functionality is typical for the development of LuaTₑX and experimental math fonts and no user should ever be bothered with it. However, it demonstrates that we're not stuck with only features built in the fonts.

It can be that a user is not satisfied by some aspects of a math font design. There is not much that we can do about the shapes, but we can manipulate for instance dimensions.

For this there are two mechanism available: automatically applied dimensional fixes and a `mathdimensions` feature. Both work with the same goody specification.

```
mathematics = {
  ...
  dimensions = {
  },
  ...
}
```

The entries in a dimensions table are tables themselves. There can be many of them so one can organize dimensional tweaks in groups. The `default` group is always applied, while others are applied on demand. Say that want to tweak all ± and ∓.[7]

```
mathematics = {
```

---

[7] In fact, this mechanism is a a response to a mail on the ConTₑXt mailing list.

```
    dimensions = {
      default = {
        [0x00B1] = { -- ±
          height = 500,
          depth  = 0,
        },
        [0x2213] = { -- ∓
          height = 500,
          depth  = 0,
        },
      },
    },
}
```

This will give these two characters a different height and depth. However, this will not have much effect in rendering (much larger dimensions might have).

```
mathematics = {
  dimensions = {
    default = {
      [0x00B1] = { -- ±
        yoffset =  100,
      },
      [0x2213] = { -- ∓
        yoffset = -100,
      },
    },
  },
}
```

This will raise and lower the glyphs in their bounding boxes and give them an appearance more close to their ancestors. But defined this way, they are always applied and that might not be what we want. So, we can do this:

```
mathematics = {
  dimensions = {
    signs = {
      [0x00B1] = { -- ±
        yoffset =  100,
      },
      [0x2213] = { -- ∓
        yoffset = -100,
      },
    },
  },
}
```

This time the application is feature driven. As with all features, setting them up has to happen *before* fonts are loaded. This will do the trick:

```
\definefontfeature [lm-math] [mathdimensions=signs]
```

The lm-math feature is not defined by default but can be used for such purposes. It *is* defined with the fontname:

```
\definefontsynonym
  [LMMathRoman-Regular]
  [file:latinmodern-math-regular.otf]
  [features={math\mathsizesuffix,lm-math},
   goodies=lm]
```

A rather specialized goodie is the one that is used to specify math cut-ins. A good quality math font has these kerns already defined but even then you might want to add or replace some by your own. Here is an example of such a patch. Normally there are multiple goodies defined in one file but we only show kerns here:

```
local kern_200 = { bottomright = { { kern = -200 } } }
local kern_100 = { bottomright = { { kern = -100 } } }

return {
    name = "pagella-math",
    version = "1.00",
    comment = "Goodies that complement pagella.",
    author = "Hans Hagen",
    copyright = "ConTeXt development team",
    mathematics = {
        kerns = {
            [0x1D449] = kern_200, -- math italic V
            [0x1D44A] = kern_100, -- math italic W
        },
    },
}
```

As with other goodies the file is loaded with:

```
\definefontsynonym
  [MathRoman]                      % names used in definitions
  [file:texgyrepagella-math.otf]   % the file to be loaded
  [features=math\mathsizesuffix,   % size dependent features
   goodies=pagella-math]           % the goodie file to be applied
```

This is typically a goodie that is always applied and not driven by a feature. After all, the values given are passed to the engine (after being scaled).

Most goodies are bound to fonts of collections of fonts. This is different for treatments. These ship with the distribution but you can also provide your own. As this is still somewhat experimental we just mention a few aspects. The entries are filenames that point to tables.

```
return {
  name = "treatments",
  version = "1.00",
  comment = "Goodies that deals with some general issues.",
  author = "Hans Hagen",
  copyright = "ConTeXt development team",
  treatments = {
    ["adobeheitistd-regular.otf"] = {
      embedded = false, -- not yet used
      comment  = "this font is part of acrobat",
    },
    ["crap.ttf"] = {
      ignored = true,
      comment = "a text file with suffix ttf",
    },
    ["latinmodern-math.otf"] = {
      comment = "experimental",
    },
    ["rubish-regular.ttf"] = {
      comment = "check output for missing à and á",
    }
  },
}
```

The comment entry in such a table becomes part of the message at the end of a run:

```
mkiv lua stats  > loaded fonts: 2 files: latinmodern-math.otf (experimental), lmroman12-regular.otf
```

The ignored flag signals the font name database builder to ignore the file. This means that the font can still be known as file, but that its (name based) properties are not collected. As you asked explicitly for a file, the file can still be loaded. You can use this trick to avoid issues with the database builder in case of a problematic file, but a real run will still try to load the file. After all, you get what you ask for. If loading and usage is successful you get at least the message reported at the end of the run.

5.5 Analyzers

An OpenType font is kind of special in the sense that it provides some information on how to turn sequences of characters into sequences of glyphs. In fact, if all fonts had a reasonable repertoire of glyphs most of the information that concerns combining, remapping and shuffling the input and/or mapping onto glyphs could as well happen in the

renderer. This means that fonts have many of their internal features tables in common, or more precisely could share many gsub related issues, if only there had been some predefined sets of substitutional features.

So, for most of the time, a feature processor just does what the font demands and the font provides the information. There are however a few cases where font only provide part of the logic. Take for instance the init, medi, fina and isol features that relate to positions in the word: the start, the end, in the middle or isolated. For these features to work the engine has to provide information about the state of a character (glyph) and this is where analysis kicks in. Just watch this:

```
\definefontfeature
  [default-with-analyze]
  [default]
  [script=latn,mode=node,
   init=yes,medi=yes,fina=yes,isol=yes]

\showotfcomposition
  {dejavu-serif*default-with-analyze at 24pt}
  {}
  {I don't wanna know tha\utfchar{"300}t!}
```

In the tracer the different categories are colored. This kind of information is especially important for typesetting Arabic. Normally ConTEXt can figure out itself when this is needed so you don't have to worry too much about this kind of additional actions.

font 95: DejaVuSerif.ttf @ 24.0pt

features analyze=yes, autolanguage=position, autoscript=position,
checkmarks=yes, curs=yes, devanagari=yes, dummies=yes,
extensions=yes, extrafeatures=yes, extraprivates=yes,
fina=yes, init=yes, isol=yes, kern=yes, liga=yes, mark=yes,
mathkerns=yes, mathrules=yes, medi=yes, mkmk=yes, mode=node,
script=latn, spacekern=yes, tlig=yes, trep=yes

step 1

U+49:I [glue] U+64:d U+6F:o U+6E:n U+27:' U+74:t [glue]
U+77:w U+61:a U+6E:n U+6E:n U+61:a [glue] U+6B:k
U+6E:n U+6F:o U+77:w [glue] U+74:t U+68:h U+61:a U+300:`
U+74:t U+21:!

feature 'trep', type 'gsub_single', lookup 'trep', replacing
U+00027 (quotesingle) by single U+02019 (quoteright)

step 2 I don't wanna know that!

U+49:I [glue] U+64:d U+6F:o U+6E:n U+2019:' U+74:t [glue]
U+77:w U+61:a U+6E:n U+6E:n U+61:a [glue] U+6B:k
U+6E:n U+6F:o U+77:w [glue] U+74:t U+68:h U+61:a U+300:`
U+74:t U+21:!

```
feature 'mark', type 'gpos_mark2base', lookup 'p_s_1', anchor ,
  bound 1, anchoring mark U+00300 (gravecomb) to basechar
  U+00061 (a) => (1.64063pt,0pt)
```

result I don't wanna know that!

U+49:I [glue] U+64:d U+6F:o U+6E:n U+2019:' U+74:t [glue]
U+77:w U+61:a U+6E:n U+6E:n U+61:a [glue] U+6B:k
U+6E:n U+6F:o U+77:w [glue] U+74:t U+68:h U+61:a U+300:`
U+74:t U+21:!

5.6 Processors

Todo.

5.7 Optimizing

Todo.

5.8 Tracing

There are a lot of tracing options in MkIV, but most will never be seen by users. Most are enabled using the tracker mechanism. Some have a bit more visibility and have a dedicated command to trigger them.

When something is going terribly wrong, you will always get a message but sometimes even an end-user has to request for more information. An example are missing characters. There are several ways to get them reported:

```
\enabletrackers[fonts.missing=replace]
\enabletrackers[fonts.missing=remove]
\enabletrackers[fonts.missing]
```

For historic reasons we also have:

```
\checkcharactersinfont
\removemissingcharacters
\replacemissingcharacters
```

which happens automatically when you enable the tracker. There is some extra overhead involved so you might want to turn on this feature on only if you really expect characters not to be present.

Say that we use Latin Modern fonts and ask for some of the rare fractions:

```
\definedfont[lmroman10-regular*default-with-missing at 10pt]
a b c ½ ⅓ ¼ ⅕ ⅙ ⅛ Ɣ ɣ ʤ ʭ ɥ α β γ
```

We get this: a b c ½ □ ¼ □ □ □ □ □ □ □ □ □ □ □. In the log file you will find something like this:

```
fonts   > characters > start missing characters: lmroman10-regular.otf

missing > U+00194   Ɣ   LATIN CAPITAL LETTER GAMMA
missing > U+00263   ɣ   LATIN SMALL LETTER GAMMA
missing > U+002A4   ʤ   LATIN SMALL LETTER DEZH DIGRAPH
missing > U+002AD   ʭ   LATIN LETTER BIDENTAL PERCUSSIVE
missing > U+002AE   ɥ   LATIN SMALL LETTER TURNED H WITH FISHHOOK
missing > U+003B1   α   GREEK SMALL LETTER ALPHA
missing > U+003B2   β   GREEK SMALL LETTER BETA
missing > U+003B3   γ   GREEK SMALL LETTER GAMMA
missing > U+02153   ⅓   VULGAR FRACTION ONE THIRD
missing > U+02155   ⅕   VULGAR FRACTION ONE FIFTH
missing > U+02159   ⅙   VULGAR FRACTION ONE SIXTH
missing > U+0215B   ⅛   VULGAR FRACTION ONE EIGHTH

fonts   > characters > stop missing characters
```

If you're lucky your editor will use a font that shows the missing characters (dejavu monospace is a good candidate).

The replacement characters can help you to locate the spots where something is missing so that an alternative can be considered. The replacements resemble the category of the missing character.

symbol	name
□	placeholder lowercase blue
□	placeholder lowercase magenta
□	placeholder lowercase red
□	placeholder lowercase yellow
'	placeholder mark green

```
▌      placeholder punctuation cyan
▯      placeholder uppercase red
```

You can call up this legend after loading an extra module:

```
\usemodule[s][fonts-missing]
```

```
\showmissingcharacterslegend
```

```
\showmissingcharacters
```

The last command shows a detailed list of missing characters

filename lmroman10-regular.otf
missing   12

```
U+00194  Ɣ  LATIN CAPITAL LETTER GAMMA
U+00263  ɣ  LATIN SMALL LETTER GAMMA
U+002A4  ʤ  LATIN SMALL LETTER DEZH DIGRAPH
U+002AD  ʭ  LATIN LETTER BIDENTAL PERCUSSIVE
U+002AE  ʮ  LATIN SMALL LETTER TURNED H WITH FISHHOOK
U+003B1  α  GREEK SMALL LETTER ALPHA
U+003B2  β  GREEK SMALL LETTER BETA
U+003B3  γ  GREEK SMALL LETTER GAMMA
U+02153  ⅓  VULGAR FRACTION ONE THIRD
U+02155  ⅕  VULGAR FRACTION ONE FIFTH
U+02159  ⅙  VULGAR FRACTION ONE SIXTH
U+0215B  ⅛  VULGAR FRACTION ONE EIGHTH
```

Here the characters are shown, because we use a monospaced font that happens to have them. Of course this example uses characters that are rarely used and are unlikely to show up in future versions of the Latin Modern fonts.

Here a few more relevant trackers will be mentioned.

5.9 Discretionaries

The font handler has to do some magick to get features working with and across discretionaries. To some extend you can use font switches inside discretionaries but for sure border cases are not dealt with. This works:

12567	123	12567	123
	467		467
12₅67	12₃	12₅67	12₃
	467		467
125₅567	123₃	125₅567	123₃
	₄467		₄467

| 1 | 2 | 3 | 4 |

The first two examples have `otf.alwaysdisk` enabled, the last two have it disabled.

```
\definedfont[cambria*default]
12\discretionary
    {3} {4} {5}%
67\par
12{\oldstyle\discretionary
    {3} {4} {5}}%
67\par
12\discretionary
    {3{\oldstyle3}} {{\oldstyle4}4} {5{\oldstyle5}5}%
67\par
```

5.10 Some remarks

If you talk about features and fonts it is not difficult to end up speaking OpenType. However, in ConTEXt we use the term in a more general way, if only because we provide more features. In traditional TEX we have a few features: ligatures, smallcaps and kerns, and to some extent we can see oldstyle numerals also as feature. It is however important to notice that in OpenType ligatures are just a synonym for combining multiple characters into on. From the user interface point of view these operations are grouped into `liga`, `dlig`, `clig` and `rlig` and for TEXies we have `tlig`. The distinction is not as clear as one might think: any feature can use the ligature builder. And as a consequence we see that happen too, for instance some fonts use `ccmp` for constructing mandatory ligatures.

Some of these interpretations (or maybe even tricks) are side effects of for instance user interfaces. If one can for instance not turn on or off the `ccmp` feature, but can do that for `liga`, then one way to keep some ligatures in for instance letter spaced text, is to put them into `ccmp`, assuming that this one will always be enabled. Eventually that then becomes a sort of standard. Personally I don't like such pseudo standards but we have to live with them.

Another example of such a standard is the used of non breakable spaces to influence treatment of some Devanagari characters. Where Unicode has special characters to influence mechanisms that combine and replace characters, the lack of some triggers others to be used and eventually that becomes a standard. Similar ambiguities arise with

math: we have no way to indicate math (while we do have ways to indicate a change in writing order).

Talking of math, take OpenType math: at some point there is a draft, that then gets implemented in one word processor using one font, but omissions or imperfections that surface (maybe because more fonts and engines are developed) stay around because the initial implementation is published and frozen, simply because there are many users that stick to expectations. Where T_EXies accept a few years of development, this is not true for commercial applications.[8]

So, although there is without doubt progress, some annoyances stay. The T_EX community has always been able to adapt, and this is one reason why a Lua implementation is nice: it gives room for experiments, extensions, variants, etc. Of course it also makes a bit more independent, although one may wonder if that matters any longer in a rapidly changing world. The original idea behind T_EX, that it should be useable for ages, will survive, but users might see more changes in a lifetime than foreseen when T_EX showed up.

---

[8] Of course html is the biggest example of this: we're stuck forever with open tags without close tags, mixed uppercase and lowercase tags, attributes without value or values without quotes.

6 Scripts

6.1 Introduction

As OpenType fonts can provide specific features to deal with scripts and languages it makes sense to spend some word on this.

7 Math

7.1 Introduction

As one can expect, math support in ConTEXt is to some extend modelled after what plain TEX provides, plus what was later decided to be standard. This mostly concerns the way fonts behave and what names are used to access glyphs or special constructs. It means that when you come from another macro package you can stick to coding math the way you did before, at least the basic coding. In addition to this, ConTEXt gives control over fonts, structure and rendering and most of that was either driven by personal need or user demand. To be honest, many of the options are probably of not much interest to the average user.

As we focus on fonts we will only touch this aspect of math here. Right from when we started with developing LuaTEX, cleaning op the math part of ConTEXt was part of the game. Some primitives were added that would make it possible to avoid unnecessary complex macros to get certain glyphs rendered, like radicals, accents and extensibles. This was made easy because we also support OpenType math and because we knew that eventually the Latin Modern and Gyre fonts would also support OpenType. In order to move forward and get rid of traditional eight bit fonts ConTEXt MkIV can construct a virtual OpenType font from traditional math fonts. It makes not much sense to discuss that here as by now this method is only provided for reasons of compatibility and a reference to the past. As a lot of time went into this it will always stay around if only to remind us of what we went through to get where we are now.

7.2 Unicode math

Due to the limited amount of glyphs in a Type1 font a macro package has to jump through loops in order to get traditional TEX engines behave well. As a practical consequence these fonts are often a mixture of characters, symbols, punctuation and snippets that make up larger shapes. The font dimensions in these files have often special meanings too.

This has all changed considerably with math being part of Unicode. It was however Microsoft where the real action took place: the development of the first font that combined Unicode with OpenType technology. The Cambria font can be considered the benchmark for fonts that surfaced later. The characteristic of a math font are the following:

- All math alphabets are present: latin as well as greek, in regular, italic, bold, fraktur and script variants as well as some combinations of these.
- The symbols that make sense are present (read: the more obscure shapes can be omited).
- For the characters that make sense, there are two variants that render well at smaller sizes: script and scriptscript. In the font they have the same size but the application will scale them down. This feature is named `ssty`.

- Characters that can extend horizontally (for instance accents and arrows) or vertically (like radicals and fences) have associated larger variants and carries information about how to grow indefinitely.
- There is a whole lot of special math dimensions. Most of the ones already used in TeX are present.
- Some glyphs come in variants in order to please special usage. There can also be variants for script or fraktur alphabets.

This means that in practice an OpenType math font is quite large. We easily have thousands of glyphs. It also means that creating such a font involves some expertise and this is one of the reasons why TeX usergroups have joined forces in developing a suite of fonts. There are also other initiatives in the TeX community, of which Xits is an example.[9] The well known Lucida Bright math font package has also been upgraded to a set of OpenType math fonts.

The fact that there are not that many math fonts out there has a positive side as well: ConTeXt comes with them pre-configured. Because during the development of LuaTeX we needed to have at least a couple of fonts for testing, and because it makes no sense to waste time on traditional fonts, the Latin Modern, Palatino, Times and a few more fonts were (and still are) provided as virtual Unicode fonts.

In a regular text font, what you key in is what you get out. So, when you've chosen a font with an italic shape, you get italic shapes, even if the smallcaps feature is enabled. In math, if you use the right unicodes you also get the shape you expect. Because in this case italic shapes are situated in one of the alphabets you explicitly choose a rendering. You can enter the right codepoints directly, so for instance if you enter Unicode character U+1D434 you will get A. In practice something like $\bi A$ should also give that character if only because that is what we have been doing for over three decades. This means that the engine has to map a regular A onto the bold italic alphabet. In a traditional approach you will use math families for this, but in ConTeXt MkIV we simply use one font and one family and let the MkIV machinery do the rest.

In text mode we switch fonts styles in the following way:

```
regular {\it italic} {\bf bold} {\bi bold italic} and so on
```

The three commands shown here are shortcuts for font switches. This input is converted into an internal representation and after whatever manipulations are applied end up as:

regular *italic* **bold** ***bold italic*** and so on

If we look at what fonts we end up with we get:

regular *italic* **bold** ***bold italic*** and so on

---

[9] This is a useable variant of Stix fonts with proper math features, some extra glyphs and experimental right–to–left shapes.

Now lets do the same in math mode:

```
$regular {\it italic} {\bf bold} {\bi bold italic} and so on$
```

This time we get a different result:

*regularitalic**boldbolditalic**andsoon*

If again we analyze the fonts you see this:

*regularitalic**boldbolditalic**andsoon*

All glyphs come from the same font. Instead of regular we get italic simply because math characters are italic by nature. The two character style switches are not really font switches but just make sure that the given input is mapped onto the right alphabet.

A traditional approach using Type1 fonts is to use a so called math family for each alphabet. In that case each alphabet maps one–to–one onto the font: when we switch to a bold family we just take the glyph that sits in that slot. In MkIV we have all characters in one family so behind the screens a given character is remapped.

Now take a look at the following example:

```
$text^{script^{scriptscript}}$
```

This renders to this, with the characters marked by font:

$text^{script^{scriptscript}}$

This time we have three different fonts: one for each of the three math sizes. But this representation is not entirely honest, because indeed we have three font instances for math, but the glyphs come from the same OpenType math font. We just load the same font three times, once for each size. In fact we load the font once, but use three copies, scaled accordingly to the relative scale the font prescribes.

There is a whole bunch of commands to choose specific characters in math mode using a regular input. These are state switching commands.

\mr	a	A
\mathdefault	a	A
\mathscript	a	$\mathcal{A}$
\mathfraktur	$\mathfrak{a}$	$\mathfrak{A}$
\mathblackboard	$\mathbb{a}$	$\mathbb{A}$
\rm \mathrm	a	A
\ss \mathss	a	A
\tt \mathtt	a	A
\tf \mathtf \tfmath	a	A

```
\sl \mathsl \slmath   a   A
\it \mathit \itmath   a   A

\bf \mathbf \bfmath   a   A
\bs \mathbs \bsmath   a   A
\bi \mathbi \bimath   a   A
```

As you can see here, some commands have synonyms. The short commands adapt themselves to text and mathmode, the longer ones are meant for use in math mode only.

In text mode distinctive shapes are either a font property (the whole font looks that way) or a stylistic alternate (an extra feature of a font). In math mode we can have alternates, but in addition to the previously mentioned alphabet switchers we have a few more:

```
\frak        a   𝔄
\cal         a   𝒜
\bbd         a   𝔸
\blackboard  a   𝔸
\fraktur     a   𝔄
\gothic      a   𝔄
```

This chapter is not meant as an introduction to math but it is good to know that math font support in ConTeXt is rather flexible. There are several mechanisms for remapping and converting characters and sequences into others and more is possible. Here is one:

```
\startformula
\reals {\mathbf R} \utfchar{"0211D} \utfchar{"1D411}
\stopformula
```

$$\mathbb{R}\mathbf{R}\mathbb{R}\mathbf{R}$$

Compare this to:

```
\setupmathematics[symbolset=blackboard-to-bold]
\startformula
\reals {\mathbf R} \utfchar{"0211D} \utfchar{"1D411}
\stopformula
```

$$\mathbf{R}\mathbf{R}\mathbf{R}\mathbf{R}$$

Greek is always troublesome because instead of regular text shapes math uses a few variants. Because in Unicode characters are only included once, we have gaps in the math alphabets but MkIV will take care of this.[10] Depending on the field an author has to choose between upright and italic greek:

```
$\nabla \alpha \mathgreekupright \nabla \alpha \mathgreekitalic \nabla \alpha$
```

---

[10] This is a typical example of where exceptions in a standard force all applications that deal with it have to implement tweaks.

$$\nabla\alpha\nabla\alpha\nabla\alpha$$

By default ConTEXt is set up as follows:

```
\setupmathematics
  [sygreek=normal,
   lcgreek=italic,
   ucgreek=normal]
```

Again, these are not features of a font. The font just provides the glyphs and the TEX engine, controlled by ConTEXt takes care of mapping characters to glyphs and building special constructs. The same is true for spacing. Although math fonts do have kerning information, most of the math spacing is controlled by properties of characters and not by the font.

`$a \mathord{+} b$`	$a+b$
`$a \mathpunct{+} b$`	$a+b$
`$a \mathinner{+} b$`	$a+b$
`$a \mathop{+} b$`	$a+b$
`$a \mathalpha{+} b$`	$a+b$
`$a \mathnothing{+} b$`	$a+b$
`$a \mathbin{+} b$`	$a+b$
`$a \mathrel{+} b$`	$a+b$

As a user you don't have to worry about these issues because characters are tagged according to their usage.[11]

With TEX being the oldest and still dominant math renderer it is no surprise that Microsoft modelled its math renderer after TEX and Cambria quite well suits the concept. In retrospect it is somewhat unfortunate that we're still stuck with some left overs (or compromises) from the past with respect to spacing built into the font. However, as long as this is consistent over fonts it's not that relevant. You can always influence the spacing with the commands mentioned.

If you look at the low level definitions in for instance plain TEX but also in ConTEXt MkII that relate to prime symbols it probably takes a while before you figure out what happens there. For instance, the prime symbol is triggered by a quote and multiple in a row results in primes that are spaced tightly. In Unicode we have slots for single, double and tripple primes. Therefore, in MkIV we have a mechanism that accepts different kinds of input that eventually all end up in one of these three glyphs.

`$f^2$`	f^2
`$f\prime ^2$`	f'^2
`$f\prime \prime ^2$`	f''^2
`$f\prime \prime \prime ^2$`	f'''^2

---

[11] There are a few more commands, like `\mathlimop`, `\mathnolop` and `mathbox` but these are used differently.

```
$f{\prime }^2$                                              f′2
$f{\prime \prime }^2$                                       f″2
$f{\prime \prime \prime }^2$                                f‴2
$f'(x)$                                                     f′(x)
$f''(x)$                                                    f″(x)
$f'''(x)$                                                   f‴(x)
$f\utfchar{0x2032}(x)$                                      ƒx2032(x)
$f\utfchar{0x2033}(x)$                                      ƒx2033(x)
$f\utfchar{0x2034}(x)$                                      ƒx2034(x)
$f\utfchar{0x2032}\utfchar{0x2032}(x)$                      ƒx2032x2032(x)
$f\utfchar{0x2032}\utfchar{0x2032}\utfchar{0x2032}(x)$      ƒx2032x2032x2032(x)
$f\utfchar{0x2033}\utfchar{0x2032}(x)$                      ƒx2033x2032(x)
$f\utfchar{0x2032}\utfchar{0x2033}(x)$                      ƒx2032x2033(x)
```

Again, this is not the same as ligature building features in text fonts, but handled in a different way.

The TeX engine understands the concept of italic correction. When an italic shape is followed by for instance an upright shape, you can insert a \/ and the engine will add a correction as defined in the font. In OpenType we don't have such corrections available but we can fake it, which is what the itlc feature in ConTeXt does. However, you need to enable this feature explicitly. An example of a setup is:

```
\definefontfeature
  [default]
  [default]
  [itlc=yes,textitalics=yes]

\setupitaliccorrection
  [global,always]
```

This will make sure that the right amount of correction is added between italic shapes and non italics or boxes. Using text instead of always would limit the correction to glyphs only and leaving out the global would permit selective (grouped) usage at the cost of more runtime. There is no need for the \/ here.

In math we also can have italic correction but there it is built into the engine and in traditional TeX no directives are needed. Italic correction properties in math fonts are somewhat troublesome as their application depends on what we're dealing with: symbols, super- and subscripts, etc. Because early versions of LuaTeX didn't handle all of it well, if only because the fonts were not yet okay, the MkIV math handler provides a bit of control.

0 *m t* *m t* *t m t m t*   no correction
1 *m t* *m t* *t m t m t*   only apply italics when the font carries them
2 *m t* *m t* *t m t m t*   apply italics provided by the font or automatically calculated

3 `mt` `mt` `tmtmt` apply italics based on an emwidth and character properties
4 `mt` `mt` `tmtmt` use method 1 but fall back on 3 if needed

Because we cannot rely on fonts too much, we default to method 3 which in practice works out well, so the setup is:

```
\setupmathematics
  [italics=3]
```

There are all kind of commands that can be used to build math constructs in such a way that super- and subscripts are consistently rendered. It goes beyond this chapter to discuss them and most users will never see or use those commands. The main message of the examples above is that text and math use different fonts and properties and therefore also different methods in rendering text or a formula. Even if the names of mechanisms are the same (like italics) you cannot assume that both modes do exactly the same.

7.3 Bold math

If you look at what Unicode provides you will notice that there are quite some bold characters. First of all there are a bunch of alphabets and because bold is not present in the text part of Unicode these alphabets have no holes. Then there are some symbols that have special meaning.

U+04X **A** MATHEMATICAL BOLD CAPITAL A
U+04X **B** MATHEMATICAL BOLD CAPITAL B
U+04X **C** MATHEMATICAL BOLD CAPITAL C
U+04X **D** MATHEMATICAL BOLD CAPITAL D
U+04X **E** MATHEMATICAL BOLD CAPITAL E
U+04X **F** MATHEMATICAL BOLD CAPITAL F
U+04X **G** MATHEMATICAL BOLD CAPITAL G
U+04X **H** MATHEMATICAL BOLD CAPITAL H
U+04X **I** MATHEMATICAL BOLD CAPITAL I
U+04X **J** MATHEMATICAL BOLD CAPITAL J
U+04X **K** MATHEMATICAL BOLD CAPITAL K
U+04X **L** MATHEMATICAL BOLD CAPITAL L
U+04X **M** MATHEMATICAL BOLD CAPITAL M
U+04X **N** MATHEMATICAL BOLD CAPITAL N
U+04X **O** MATHEMATICAL BOLD CAPITAL O
U+04X **P** MATHEMATICAL BOLD CAPITAL P
U+04X **Q** MATHEMATICAL BOLD CAPITAL Q
U+04X **R** MATHEMATICAL BOLD CAPITAL R
U+04X **S** MATHEMATICAL BOLD CAPITAL S
U+04X **T** MATHEMATICAL BOLD CAPITAL T
U+04X **U** MATHEMATICAL BOLD CAPITAL U
U+04X **V** MATHEMATICAL BOLD CAPITAL V

U+04X	**W**	MATHEMATICAL BOLD CAPITAL W
U+04X	**X**	MATHEMATICAL BOLD CAPITAL X
U+04X	**Y**	MATHEMATICAL BOLD CAPITAL Y
U+04X	**Z**	MATHEMATICAL BOLD CAPITAL Z
U+04X	**a**	MATHEMATICAL BOLD SMALL A
U+04X	**b**	MATHEMATICAL BOLD SMALL B
U+04X	**c**	MATHEMATICAL BOLD SMALL C
U+04X	**d**	MATHEMATICAL BOLD SMALL D
U+04X	**e**	MATHEMATICAL BOLD SMALL E
U+04X	**f**	MATHEMATICAL BOLD SMALL F
U+04X	**g**	MATHEMATICAL BOLD SMALL G
U+04X	**h**	MATHEMATICAL BOLD SMALL H
U+04X	**i**	MATHEMATICAL BOLD SMALL I
U+04X	**j**	MATHEMATICAL BOLD SMALL J
U+04X	**k**	MATHEMATICAL BOLD SMALL K
U+04X	**l**	MATHEMATICAL BOLD SMALL L
U+04X	**m**	MATHEMATICAL BOLD SMALL M
U+04X	**n**	MATHEMATICAL BOLD SMALL N
U+04X	**o**	MATHEMATICAL BOLD SMALL O
U+04X	**p**	MATHEMATICAL BOLD SMALL P
U+04X	**q**	MATHEMATICAL BOLD SMALL Q
U+04X	**r**	MATHEMATICAL BOLD SMALL R
U+04X	**s**	MATHEMATICAL BOLD SMALL S
U+04X	**t**	MATHEMATICAL BOLD SMALL T
U+04X	**u**	MATHEMATICAL BOLD SMALL U
U+04X	**v**	MATHEMATICAL BOLD SMALL V
U+04X	**w**	MATHEMATICAL BOLD SMALL W
U+04X	**x**	MATHEMATICAL BOLD SMALL X
U+04X	**y**	MATHEMATICAL BOLD SMALL Y
U+04X	**z**	MATHEMATICAL BOLD SMALL Z
U+04X	*A*	MATHEMATICAL BOLD ITALIC CAPITAL A
U+04X	*B*	MATHEMATICAL BOLD ITALIC CAPITAL B
U+04X	*C*	MATHEMATICAL BOLD ITALIC CAPITAL C
U+04X	*D*	MATHEMATICAL BOLD ITALIC CAPITAL D
U+04X	*E*	MATHEMATICAL BOLD ITALIC CAPITAL E
U+04X	*F*	MATHEMATICAL BOLD ITALIC CAPITAL F
U+04X	*G*	MATHEMATICAL BOLD ITALIC CAPITAL G
U+04X	*H*	MATHEMATICAL BOLD ITALIC CAPITAL H
U+04X	*I*	MATHEMATICAL BOLD ITALIC CAPITAL I
U+04X	*J*	MATHEMATICAL BOLD ITALIC CAPITAL J
U+04X	*K*	MATHEMATICAL BOLD ITALIC CAPITAL K
U+04X	*L*	MATHEMATICAL BOLD ITALIC CAPITAL L
U+04X	*M*	MATHEMATICAL BOLD ITALIC CAPITAL M
U+04X	*N*	MATHEMATICAL BOLD ITALIC CAPITAL N

U+04X	*O*	MATHEMATICAL BOLD ITALIC CAPITAL O
U+04X	*P*	MATHEMATICAL BOLD ITALIC CAPITAL P
U+04X	*Q*	MATHEMATICAL BOLD ITALIC CAPITAL Q
U+04X	*R*	MATHEMATICAL BOLD ITALIC CAPITAL R
U+04X	*S*	MATHEMATICAL BOLD ITALIC CAPITAL S
U+04X	*T*	MATHEMATICAL BOLD ITALIC CAPITAL T
U+04X	*U*	MATHEMATICAL BOLD ITALIC CAPITAL U
U+04X	*V*	MATHEMATICAL BOLD ITALIC CAPITAL V
U+04X	*W*	MATHEMATICAL BOLD ITALIC CAPITAL W
U+04X	*X*	MATHEMATICAL BOLD ITALIC CAPITAL X
U+04X	*Y*	MATHEMATICAL BOLD ITALIC CAPITAL Y
U+04X	*Z*	MATHEMATICAL BOLD ITALIC CAPITAL Z
U+04X	*a*	MATHEMATICAL BOLD ITALIC SMALL A
U+04X	*b*	MATHEMATICAL BOLD ITALIC SMALL B
U+04X	*c*	MATHEMATICAL BOLD ITALIC SMALL C
U+04X	*d*	MATHEMATICAL BOLD ITALIC SMALL D
U+04X	*e*	MATHEMATICAL BOLD ITALIC SMALL E
U+04X	*f*	MATHEMATICAL BOLD ITALIC SMALL F
U+04X	*g*	MATHEMATICAL BOLD ITALIC SMALL G
U+04X	*h*	MATHEMATICAL BOLD ITALIC SMALL H
U+04X	*i*	MATHEMATICAL BOLD ITALIC SMALL I
U+04X	*j*	MATHEMATICAL BOLD ITALIC SMALL J
U+04X	*k*	MATHEMATICAL BOLD ITALIC SMALL K
U+04X	*l*	MATHEMATICAL BOLD ITALIC SMALL L
U+04X	*m*	MATHEMATICAL BOLD ITALIC SMALL M
U+04X	*n*	MATHEMATICAL BOLD ITALIC SMALL N
U+04X	*o*	MATHEMATICAL BOLD ITALIC SMALL O
U+04X	*p*	MATHEMATICAL BOLD ITALIC SMALL P
U+04X	*q*	MATHEMATICAL BOLD ITALIC SMALL Q
U+04X	*r*	MATHEMATICAL BOLD ITALIC SMALL R
U+04X	*s*	MATHEMATICAL BOLD ITALIC SMALL S
U+04X	*t*	MATHEMATICAL BOLD ITALIC SMALL T
U+04X	*u*	MATHEMATICAL BOLD ITALIC SMALL U
U+04X	*v*	MATHEMATICAL BOLD ITALIC SMALL V
U+04X	*w*	MATHEMATICAL BOLD ITALIC SMALL W
U+04X	*x*	MATHEMATICAL BOLD ITALIC SMALL X
U+04X	*y*	MATHEMATICAL BOLD ITALIC SMALL Y
U+04X	*z*	MATHEMATICAL BOLD ITALIC SMALL Z
U+04X	𝓐	MATHEMATICAL BOLD SCRIPT CAPITAL A
U+04X	𝓑	MATHEMATICAL BOLD SCRIPT CAPITAL B
U+04X	𝓒	MATHEMATICAL BOLD SCRIPT CAPITAL C
U+04X	𝓓	MATHEMATICAL BOLD SCRIPT CAPITAL D
U+04X	𝓔	MATHEMATICAL BOLD SCRIPT CAPITAL E
U+04X	𝓕	MATHEMATICAL BOLD SCRIPT CAPITAL F

U+04X	𝓖	MATHEMATICAL BOLD SCRIPT CAPITAL G
U+04X	𝓗	MATHEMATICAL BOLD SCRIPT CAPITAL H
U+04X	𝓘	MATHEMATICAL BOLD SCRIPT CAPITAL I
U+04X	𝓙	MATHEMATICAL BOLD SCRIPT CAPITAL J
U+04X	𝓚	MATHEMATICAL BOLD SCRIPT CAPITAL K
U+04X	𝓛	MATHEMATICAL BOLD SCRIPT CAPITAL L
U+04X	𝓜	MATHEMATICAL BOLD SCRIPT CAPITAL M
U+04X	𝓝	MATHEMATICAL BOLD SCRIPT CAPITAL N
U+04X	𝓞	MATHEMATICAL BOLD SCRIPT CAPITAL O
U+04X	𝓟	MATHEMATICAL BOLD SCRIPT CAPITAL P
U+04X	𝓠	MATHEMATICAL BOLD SCRIPT CAPITAL Q
U+04X	𝓡	MATHEMATICAL BOLD SCRIPT CAPITAL R
U+04X	𝓢	MATHEMATICAL BOLD SCRIPT CAPITAL S
U+04X	𝓣	MATHEMATICAL BOLD SCRIPT CAPITAL T
U+04X	𝓤	MATHEMATICAL BOLD SCRIPT CAPITAL U
U+04X	𝓥	MATHEMATICAL BOLD SCRIPT CAPITAL V
U+04X	𝓦	MATHEMATICAL BOLD SCRIPT CAPITAL W
U+04X	𝓧	MATHEMATICAL BOLD SCRIPT CAPITAL X
U+04X	𝓨	MATHEMATICAL BOLD SCRIPT CAPITAL Y
U+04X	𝓩	MATHEMATICAL BOLD SCRIPT CAPITAL Z
U+04X	𝓪	MATHEMATICAL BOLD SCRIPT SMALL A
U+04X	𝓫	MATHEMATICAL BOLD SCRIPT SMALL B
U+04X	𝓬	MATHEMATICAL BOLD SCRIPT SMALL C
U+04X	𝓭	MATHEMATICAL BOLD SCRIPT SMALL D
U+04X	𝓮	MATHEMATICAL BOLD SCRIPT SMALL E
U+04X	𝓯	MATHEMATICAL BOLD SCRIPT SMALL F
U+04X	𝓰	MATHEMATICAL BOLD SCRIPT SMALL G
U+04X	𝓱	MATHEMATICAL BOLD SCRIPT SMALL H
U+04X	𝓲	MATHEMATICAL BOLD SCRIPT SMALL I
U+04X	𝓳	MATHEMATICAL BOLD SCRIPT SMALL J
U+04X	𝓴	MATHEMATICAL BOLD SCRIPT SMALL K
U+04X	𝓵	MATHEMATICAL BOLD SCRIPT SMALL L
U+04X	𝓶	MATHEMATICAL BOLD SCRIPT SMALL M
U+04X	𝓷	MATHEMATICAL BOLD SCRIPT SMALL N
U+04X	𝓸	MATHEMATICAL BOLD SCRIPT SMALL O
U+04X	𝓹	MATHEMATICAL BOLD SCRIPT SMALL P
U+04X	𝓺	MATHEMATICAL BOLD SCRIPT SMALL Q
U+04X	𝓻	MATHEMATICAL BOLD SCRIPT SMALL R
U+04X	𝓼	MATHEMATICAL BOLD SCRIPT SMALL S
U+04X	𝓽	MATHEMATICAL BOLD SCRIPT SMALL T
U+04X	𝓾	MATHEMATICAL BOLD SCRIPT SMALL U
U+04X	𝓿	MATHEMATICAL BOLD SCRIPT SMALL V
U+04X	𝔀	MATHEMATICAL BOLD SCRIPT SMALL W
U+04X	𝔁	MATHEMATICAL BOLD SCRIPT SMALL X

Math

U+04X	𝑦	MATHEMATICAL BOLD SCRIPT SMALL Y
U+04X	𝑧	MATHEMATICAL BOLD SCRIPT SMALL Z
U+04X	𝕬	MATHEMATICAL BOLD FRAKTUR CAPITAL A
U+04X	𝕭	MATHEMATICAL BOLD FRAKTUR CAPITAL B
U+04X	𝕮	MATHEMATICAL BOLD FRAKTUR CAPITAL C
U+04X	𝕯	MATHEMATICAL BOLD FRAKTUR CAPITAL D
U+04X	𝕰	MATHEMATICAL BOLD FRAKTUR CAPITAL E
U+04X	𝕱	MATHEMATICAL BOLD FRAKTUR CAPITAL F
U+04X	𝕲	MATHEMATICAL BOLD FRAKTUR CAPITAL G
U+04X	𝕳	MATHEMATICAL BOLD FRAKTUR CAPITAL H
U+04X	𝕴	MATHEMATICAL BOLD FRAKTUR CAPITAL I
U+04X	𝕵	MATHEMATICAL BOLD FRAKTUR CAPITAL J
U+04X	𝕶	MATHEMATICAL BOLD FRAKTUR CAPITAL K
U+04X	𝕷	MATHEMATICAL BOLD FRAKTUR CAPITAL L
U+04X	𝕸	MATHEMATICAL BOLD FRAKTUR CAPITAL M
U+04X	𝕹	MATHEMATICAL BOLD FRAKTUR CAPITAL N
U+04X	𝕺	MATHEMATICAL BOLD FRAKTUR CAPITAL O
U+04X	𝕻	MATHEMATICAL BOLD FRAKTUR CAPITAL P
U+04X	𝕼	MATHEMATICAL BOLD FRAKTUR CAPITAL Q
U+04X	𝕽	MATHEMATICAL BOLD FRAKTUR CAPITAL R
U+04X	𝕾	MATHEMATICAL BOLD FRAKTUR CAPITAL S
U+04X	𝕿	MATHEMATICAL BOLD FRAKTUR CAPITAL T
U+04X	𝖀	MATHEMATICAL BOLD FRAKTUR CAPITAL U
U+04X	𝖁	MATHEMATICAL BOLD FRAKTUR CAPITAL V
U+04X	𝖂	MATHEMATICAL BOLD FRAKTUR CAPITAL W
U+04X	𝖃	MATHEMATICAL BOLD FRAKTUR CAPITAL X
U+04X	𝖄	MATHEMATICAL BOLD FRAKTUR CAPITAL Y
U+04X	𝖅	MATHEMATICAL BOLD FRAKTUR CAPITAL Z
U+04X	𝖆	MATHEMATICAL BOLD FRAKTUR SMALL A
U+04X	𝖇	MATHEMATICAL BOLD FRAKTUR SMALL B
U+04X	𝖈	MATHEMATICAL BOLD FRAKTUR SMALL C
U+04X	𝖉	MATHEMATICAL BOLD FRAKTUR SMALL D
U+04X	𝖊	MATHEMATICAL BOLD FRAKTUR SMALL E
U+04X	𝖋	MATHEMATICAL BOLD FRAKTUR SMALL F
U+04X	𝖌	MATHEMATICAL BOLD FRAKTUR SMALL G
U+04X	𝖍	MATHEMATICAL BOLD FRAKTUR SMALL H
U+04X	𝖎	MATHEMATICAL BOLD FRAKTUR SMALL I
U+04X	𝖏	MATHEMATICAL BOLD FRAKTUR SMALL J
U+04X	𝖐	MATHEMATICAL BOLD FRAKTUR SMALL K
U+04X	𝖑	MATHEMATICAL BOLD FRAKTUR SMALL L
U+04X	𝖒	MATHEMATICAL BOLD FRAKTUR SMALL M
U+04X	𝖓	MATHEMATICAL BOLD FRAKTUR SMALL N
U+04X	𝖔	MATHEMATICAL BOLD FRAKTUR SMALL O
U+04X	𝖕	MATHEMATICAL BOLD FRAKTUR SMALL P

U+04X	q	MATHEMATICAL BOLD FRAKTUR SMALL Q
U+04X	r	MATHEMATICAL BOLD FRAKTUR SMALL R
U+04X	s	MATHEMATICAL BOLD FRAKTUR SMALL S
U+04X	t	MATHEMATICAL BOLD FRAKTUR SMALL T
U+04X	u	MATHEMATICAL BOLD FRAKTUR SMALL U
U+04X	v	MATHEMATICAL BOLD FRAKTUR SMALL V
U+04X	w	MATHEMATICAL BOLD FRAKTUR SMALL W
U+04X	x	MATHEMATICAL BOLD FRAKTUR SMALL X
U+04X	y	MATHEMATICAL BOLD FRAKTUR SMALL Y
U+04X	z	MATHEMATICAL BOLD FRAKTUR SMALL Z
U+04X	A	MATHEMATICAL BOLD CAPITAL ALPHA
U+04X	B	MATHEMATICAL BOLD CAPITAL BETA
U+04X	Γ	MATHEMATICAL BOLD CAPITAL GAMMA
U+04X	Δ	MATHEMATICAL BOLD CAPITAL DELTA
U+04X	E	MATHEMATICAL BOLD CAPITAL EPSILON
U+04X	Z	MATHEMATICAL BOLD CAPITAL ZETA
U+04X	H	MATHEMATICAL BOLD CAPITAL ETA
U+04X	Θ	MATHEMATICAL BOLD CAPITAL THETA
U+04X	I	MATHEMATICAL BOLD CAPITAL IOTA
U+04X	K	MATHEMATICAL BOLD CAPITAL KAPPA
U+04X	Λ	MATHEMATICAL BOLD CAPITAL LAMDA
U+04X	M	MATHEMATICAL BOLD CAPITAL MU
U+04X	N	MATHEMATICAL BOLD CAPITAL NU
U+04X	Ξ	MATHEMATICAL BOLD CAPITAL XI
U+04X	O	MATHEMATICAL BOLD CAPITAL OMICRON
U+04X	Π	MATHEMATICAL BOLD CAPITAL PI
U+04X	P	MATHEMATICAL BOLD CAPITAL RHO
U+04X	Θ	MATHEMATICAL BOLD CAPITAL THETA SYMBOL
U+04X	Σ	MATHEMATICAL BOLD CAPITAL SIGMA
U+04X	T	MATHEMATICAL BOLD CAPITAL TAU
U+04X	Υ	MATHEMATICAL BOLD CAPITAL UPSILON
U+04X	Φ	MATHEMATICAL BOLD CAPITAL PHI
U+04X	X	MATHEMATICAL BOLD CAPITAL CHI
U+04X	Ψ	MATHEMATICAL BOLD CAPITAL PSI
U+04X	Ω	MATHEMATICAL BOLD CAPITAL OMEGA
U+04X	∇	MATHEMATICAL BOLD NABLA
U+04X	α	MATHEMATICAL BOLD SMALL ALPHA
U+04X	β	MATHEMATICAL BOLD SMALL BETA
U+04X	γ	MATHEMATICAL BOLD SMALL GAMMA
U+04X	δ	MATHEMATICAL BOLD SMALL DELTA
U+04X	ε	MATHEMATICAL BOLD SMALL EPSILON
U+04X	ζ	MATHEMATICAL BOLD SMALL ZETA
U+04X	η	MATHEMATICAL BOLD SMALL ETA
U+04X	θ	MATHEMATICAL BOLD SMALL THETA

U+04X	ι	MATHEMATICAL BOLD SMALL IOTA
U+04X	κ	MATHEMATICAL BOLD SMALL KAPPA
U+04X	λ	MATHEMATICAL BOLD SMALL LAMDA
U+04X	μ	MATHEMATICAL BOLD SMALL MU
U+04X	ν	MATHEMATICAL BOLD SMALL NU
U+04X	ξ	MATHEMATICAL BOLD SMALL XI
U+04X	o	MATHEMATICAL BOLD SMALL OMICRON
U+04X	π	MATHEMATICAL BOLD SMALL PI
U+04X	ρ	MATHEMATICAL BOLD SMALL RHO
U+04X	ς	MATHEMATICAL BOLD SMALL FINAL SIGMA
U+04X	σ	MATHEMATICAL BOLD SMALL SIGMA
U+04X	τ	MATHEMATICAL BOLD SMALL TAU
U+04X	υ	MATHEMATICAL BOLD SMALL UPSILON
U+04X	φ	MATHEMATICAL BOLD SMALL PHI
U+04X	χ	MATHEMATICAL BOLD SMALL CHI
U+04X	ψ	MATHEMATICAL BOLD SMALL PSI
U+04X	ω	MATHEMATICAL BOLD SMALL OMEGA
U+04X	∂	MATHEMATICAL BOLD PARTIAL DIFFERENTIAL
U+04X	ϵ	MATHEMATICAL BOLD EPSILON SYMBOL
U+04X	ϑ	MATHEMATICAL BOLD THETA SYMBOL
U+04X	ϰ	MATHEMATICAL BOLD KAPPA SYMBOL
U+04X	ϕ	MATHEMATICAL BOLD PHI SYMBOL
U+04X	ϱ	MATHEMATICAL BOLD RHO SYMBOL
U+04X	ϖ	MATHEMATICAL BOLD PI SYMBOL
U+04X	*A*	MATHEMATICAL BOLD ITALIC CAPITAL ALPHA
U+04X	*B*	MATHEMATICAL BOLD ITALIC CAPITAL BETA
U+04X	*Γ*	MATHEMATICAL BOLD ITALIC CAPITAL GAMMA
U+04X	*Δ*	MATHEMATICAL BOLD ITALIC CAPITAL DELTA
U+04X	*E*	MATHEMATICAL BOLD ITALIC CAPITAL EPSILON
U+04X	*Z*	MATHEMATICAL BOLD ITALIC CAPITAL ZETA
U+04X	*H*	MATHEMATICAL BOLD ITALIC CAPITAL ETA
U+04X	*Θ*	MATHEMATICAL BOLD ITALIC CAPITAL THETA
U+04X	*I*	MATHEMATICAL BOLD ITALIC CAPITAL IOTA
U+04X	*K*	MATHEMATICAL BOLD ITALIC CAPITAL KAPPA
U+04X	*Λ*	MATHEMATICAL BOLD ITALIC CAPITAL LAMDA
U+04X	*M*	MATHEMATICAL BOLD ITALIC CAPITAL MU
U+04X	*N*	MATHEMATICAL BOLD ITALIC CAPITAL NU
U+04X	*Ξ*	MATHEMATICAL BOLD ITALIC CAPITAL XI
U+04X	*O*	MATHEMATICAL BOLD ITALIC CAPITAL OMICRON
U+04X	*Π*	MATHEMATICAL BOLD ITALIC CAPITAL PI
U+04X	*P*	MATHEMATICAL BOLD ITALIC CAPITAL RHO
U+04X	*Θ*	MATHEMATICAL BOLD ITALIC CAPITAL THETA SYMBOL
U+04X	*Σ*	MATHEMATICAL BOLD ITALIC CAPITAL SIGMA
U+04X	*T*	MATHEMATICAL BOLD ITALIC CAPITAL TAU

U+04X	*Υ*	MATHEMATICAL BOLD ITALIC CAPITAL UPSILON
U+04X	*Φ*	MATHEMATICAL BOLD ITALIC CAPITAL PHI
U+04X	*X*	MATHEMATICAL BOLD ITALIC CAPITAL CHI
U+04X	*Ψ*	MATHEMATICAL BOLD ITALIC CAPITAL PSI
U+04X	*Ω*	MATHEMATICAL BOLD ITALIC CAPITAL OMEGA
U+04X	*∇*	MATHEMATICAL BOLD ITALIC NABLA
U+04X	*α*	MATHEMATICAL BOLD ITALIC SMALL ALPHA
U+04X	*β*	MATHEMATICAL BOLD ITALIC SMALL BETA
U+04X	*γ*	MATHEMATICAL BOLD ITALIC SMALL GAMMA
U+04X	*δ*	MATHEMATICAL BOLD ITALIC SMALL DELTA
U+04X	*ε*	MATHEMATICAL BOLD ITALIC SMALL EPSILON
U+04X	*ζ*	MATHEMATICAL BOLD ITALIC SMALL ZETA
U+04X	*η*	MATHEMATICAL BOLD ITALIC SMALL ETA
U+04X	*θ*	MATHEMATICAL BOLD ITALIC SMALL THETA
U+04X	*ι*	MATHEMATICAL BOLD ITALIC SMALL IOTA
U+04X	*κ*	MATHEMATICAL BOLD ITALIC SMALL KAPPA
U+04X	*λ*	MATHEMATICAL BOLD ITALIC SMALL LAMDA
U+04X	*μ*	MATHEMATICAL BOLD ITALIC SMALL MU
U+04X	*ν*	MATHEMATICAL BOLD ITALIC SMALL NU
U+04X	*ξ*	MATHEMATICAL BOLD ITALIC SMALL XI
U+04X	*o*	MATHEMATICAL BOLD ITALIC SMALL OMICRON
U+04X	*π*	MATHEMATICAL BOLD ITALIC SMALL PI
U+04X	*ρ*	MATHEMATICAL BOLD ITALIC SMALL RHO
U+04X	*ς*	MATHEMATICAL BOLD ITALIC SMALL FINAL SIGMA
U+04X	*σ*	MATHEMATICAL BOLD ITALIC SMALL SIGMA
U+04X	*τ*	MATHEMATICAL BOLD ITALIC SMALL TAU
U+04X	*υ*	MATHEMATICAL BOLD ITALIC SMALL UPSILON
U+04X	*φ*	MATHEMATICAL BOLD ITALIC SMALL PHI
U+04X	*χ*	MATHEMATICAL BOLD ITALIC SMALL CHI
U+04X	*ψ*	MATHEMATICAL BOLD ITALIC SMALL PSI
U+04X	*ω*	MATHEMATICAL BOLD ITALIC SMALL OMEGA
U+04X	*∂*	MATHEMATICAL BOLD ITALIC PARTIAL DIFFERENTIAL
U+04X	*ϵ*	MATHEMATICAL BOLD ITALIC EPSILON SYMBOL
U+04X	*ϑ*	MATHEMATICAL BOLD ITALIC THETA SYMBOL
U+04X	*ϰ*	MATHEMATICAL BOLD ITALIC KAPPA SYMBOL
U+04X	*ϕ*	MATHEMATICAL BOLD ITALIC PHI SYMBOL
U+04X	*ϱ*	MATHEMATICAL BOLD ITALIC RHO SYMBOL
U+04X	*ϖ*	MATHEMATICAL BOLD ITALIC PI SYMBOL
U+04X	?	MATHEMATICAL BOLD CAPITAL DIGAMMA
U+04X	?	MATHEMATICAL BOLD SMALL DIGAMMA
U+04X	**0**	MATHEMATICAL BOLD DIGIT ZERO
U+04X	**1**	MATHEMATICAL BOLD DIGIT ONE
U+04X	**2**	MATHEMATICAL BOLD DIGIT TWO
U+04X	**3**	MATHEMATICAL BOLD DIGIT THREE

Math

U+04X	**4**	MATHEMATICAL BOLD DIGIT FOUR
U+04X	**5**	MATHEMATICAL BOLD DIGIT FIVE
U+04X	**6**	MATHEMATICAL BOLD DIGIT SIX
U+04X	**7**	MATHEMATICAL BOLD DIGIT SEVEN
U+04X	**8**	MATHEMATICAL BOLD DIGIT EIGHT
U+04X	**9**	MATHEMATICAL BOLD DIGIT NINE

The biggest mistake one can make when discussing bold math is the assumption that these bold alphabets are meant for section titles and other structural elements that need some emphasis. This is not true, in that case we would expect the whole formula to be bold and the bold symbols or variables would be even more bold. Bold math boils down to *all* math being bold. The reason why we show the list of bold characters on the previous pages is that it gives a good impression of fact that we're mostly given alphabets in an otherwise regular font.

As Latin Modern (being derived from Computer Modern) has some bold extras in MkII to some extend we do support a complete bold math switch but mixing bold formulas with regular ones has some limitations. Math typesetting consists of two phases: first the input is translated into a special list where references to fonts are not yet resolved. Instead families are used and each family has three sizes: text, script and scriptscript. In a second pass the formula is typeset and the families get translated into fonts. So, if we change the definition of a family, say math italic into bold math italic, then the definition that is actual when the second pass takes place is used.

Although LuaTeX provides for many more families and as a consequence we could have replaced the MkII mechanism with a more complete one, instead we just forgot about it and stuck to one family for regular math and another one for bold math. Okay, this is not entirely true as later on we added some more in order to deal with bidirectional typesetting.

Only a few math fonts come with a bold variant. One of the objectives of the TeXGyre math font project is to explore the possibilities of bold math companions, but such a font will probably have less coverage, simply because no real complex math will end up in for instance section titles.

When I wrote this down there were not that many math fonts that come with a real (complete) bold variant. The ConTeXt math font subsystem tries to fill this gap as good as possible by using pseudo fonts. When a typeface doesn't define a math bold variant a pseudo setup is used. When a real bold font is used, it could be that not all alphabets are supported in which case a suitable alternative is tried.

The Xits font, assembed from Stix and enhanced by Khaled Hosny, comes with a bold variant but the coverage is not complete, at least not when I wrote this paragraph. This can go unnoticed because ConTeXt tries to work around this. On the other hand, it definitely has bold properties, which can be seen from the next example. You switch between regular and bold math with the \mr and \mb commands.

```
\switchtobodyfont[xitsbidi]
```

```
$              \sqrt{x } \quad
 \mb          \sqrt{mb} \quad
 \mathupright \sqrt{u } \quad
 \mr          \sqrt{mr} \quad
 \mathupright \sqrt{u } \quad
 \mathdefault \sqrt{d }
$
```

$$\sqrt{x} \quad \sqrt{\boldsymbol{mb}} \quad \sqrt{\mathbf{u}} \quad \sqrt{mr} \quad \sqrt{\mathrm{u}} \quad \sqrt{d}$$

You can track some of what happens with:

```
\enabletrackers[math.remapping,math.families]
```

You will get some information about remapping or when it fails if fallback remapping is used. But no matter what happens with glyphs, you will notice in this example that the radical symbol is bold indeed.

7.4 Bidirectional math

There is not that much to tell about bidirectional math typesetting, simply because the fonts are still in development. However, Khaled Hosny added some support to the Xits font. Of course you need to load this font first:

```
\switchtobodyfont[xitsbidi]
```

In the previous chapter we mentioned bold math and as Xits also comes with a bold variant which means that this command loads the whole lot (which is fast enough anyway).

Easiest is to just show a few examples. When in left to right mode we get what we are accustomed to:

```
\setupmathematics[align=l2r]
```

```
\startformula
\sqrt{x^2\over 4x} \eqno(1)
\stopformula
```

```
\startformula
5 < 6 > 4
\stopformula
```

```
\startformula
5 \leq 6 \geq 7
```

```
\stopformula
```

$$\sqrt{\frac{x^2}{4x}}1)$$

$$5 < 6 > 4$$

$$5 \leq 6 \geq 7$$

However, when we go the other way, we automatically get digits converted to arabic.

```
\setupmathematics[align=r2l,bidi=yes]
```

```
\startformula
\sqrt{ف^2\over 4ب} \eqno(1)
\stopformula
```

```
\startformula
5 < 6 > 4
\stopformula
```

```
\startformula
5 \leq 6 \geq 7
\stopformula
```

$$\sqrt{\overline{\frac{٢ف}{٤}}}(1$$

$$4 < 6 > 5$$

$$7 \geq 6 \leq 5$$

You don't have to worry about how the font is set up, but not that much is needed because ConTEXt does it for you and the Xits typescripts carries the right definitions. Just to give you an idea, we show a feature definition: The magic is in the `rtlm` feature combined with `locl`.

```
\definefontfeature
  [mathematics-r2l]
  [mathematics]
  [language=ara,
   rtlm=yes,
   locl=yes]
```

Some symbols are mirrored too:

```
\setupmathematics[align=r2l,bidi=yes]
```

```
\startformula
\sum^\infty_{س=0} س^2 \eqno(2)
```

```
\stopformula
```

$$\sum_{\sf ?=0}^{\infty} {\sf ?} (2^2)$$

And of course the extensible fences are done properly too:

```
\setupmathematics[align=r2l,bidi=yes]
```

```
\startformula
\left(\root{2} \of{155}\right)
\stopformula
```

```
\startformula
\left[\int^{55}_{123} 666^3\right]
\qquad\textstyle\left[\int^{55}_{123} 666^3\right]
\stopformula
```

```
\startformula
\left\{\sum^{55}_{123} 666^3\right\}
\stopformula
```

$$\left(\overline{155}\sqrt{}^{\,2}\right)$$

$$\left[{}^3 666 \,\Big\backslash_{123}^{55}\right] \qquad \left[{}^3 666 \,\Big\backslash_{123}^{55}\right]$$

$$\left\{{}^3 666 \sum_{123}^{55}\right\}$$

The real torture test is the radical sign. A mirrored shape is used and it grows upwards as well as leftwards.

```
\setupmathematics[align=r2l,bidi=yes]
```

```
\startformula
\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{55}}}}}}}}}}
\stopformula
```
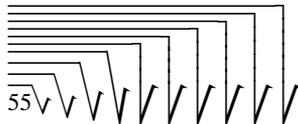

7.5 Styles

In text mode you use font switches like \sl that switches the current font to a slanted one. In math mode it is an alphabet switch in the same font. In fact, there isn't much to

choose from fonts there, apart from a massive switch to bold, in which case `\bf` is just a bolder alphabet in that bolder font.

A lot of things in math mode happen automatically. There are for instance always three instances of (the same) font active, each different in size: text, script and the smallest, scriptscript and when you ask for instance for a superscript the next smaller size is used.

normal	`\textstyle`	$text^{script^{scriptscript}}$
smaller	`\scriptstyle`	$text^{script^{scriptscript}}$
smallest	`\scriptscriptstyle`	$text^{script^{scriptscript}}$

In text style, superscripts can go twice smaller, but in script style only one smaller size is left, and in scriptscript style you're stuck with one size. The commands in the second column can be used to force a style.

The math formula builder has an important property: the formula is typeset after it has been scanned completely. In a traditional setup that has some consequences. Take this:

```
one \sl two \bf three \bi four
```

In a traditional setup four so called families are used and each character gets tagged with a family number. So we have (for instance):

$o_7 n_7 e_7 \ t_6 w_6 o_6 \ t_5 h_5 r_5 e_5 e_5 \ f_9 o_9 u_9 r_9$

As the number of families was limited there could be at most 16 families. In fact, the first four were traditionally reserved for math roman, math italic, symbol and extensibles. Then, due to the limit of 256 characters per font, another few were used for additional symbol fonts. So, adding a few more variants could exhaust the family pool quite fast. You could argue that we could halfway redefine a family but this will not work as there is a one to one relationship between family numbers and fonts assigned to them when the formula has been read in (the last value counts). And grouping won't help you either.

The actual (plain) situation is even more complex. As we have a limited number of characters per font, most symbols are accessed by name, and the name relates to a mathematical character definition using for instance `\mathchardef`. Such a definition refers to a slot in a specific family number and therefore font. It also puts a character in a so called math class. One of these, the alphanumeric class, with number 7, is special. Characters that are input directly on the keyboard (like `a–z` can also be tagged this way using `\mathcode`.

When we switch a family, this will normally not affect a symbol defined as math character, simply because we refer to a specific family/slot combination, but when a character has class 7, then it will be taken from the current family. This permits latin letters, digits and greek letters to be typeset in different styles. So, in that traditional approach we have fonts that provide a bunch of symbols as well as some alphabets. Think for instance of a font with additional symbols where the regular alphabet slots contain blackboard

shapes. The symbols are accessed directly and the characters are accessed via the regular a–z characters as these will adapt to the family and therefore font. In practice users will not notice this complication as macro packages hide the implementation details.

In MkIV the situation is different as there we have one family (or a few more if we use a full bold switch and/or bidirectional math). Although we no longer have the limit of 16 fonts we actually don't need that many families, at least not in the way we've set up MkIV[12]

```
o₁n₁e₁ t₁w₁o₁ t₁h₁r₁e₁e₁ f₁o₁u₁r₁
```

So how does this relate to styles? Each family has three fonts and we can use the switch commands to choose any of these. In text mode we use the term style for a font switch, while in math mode it's more than that: indeed we switch a font, but only in size, but the spacing is also adapted. If a proper math font is used, the smaller sizes are actually alternates in the font, visually adapted to suit their use.

In text mode we do this in order to limit the scope of a switch:

```
normal {\bf bold {\it italic} bold} normalbracket
```

This is the same as:

```
normal \bgroup \bf bold \bgroup \it italic\egroup
    \ bold\egroup \ normalbracket
```

and:

```
normal \begingroup \bf bold \begingroup \it italic\endgroup
    \ bold\endgroup \ normalbracket
```

The ConTEXt distribution ships with a plain math definition file that also uses one family but reassigns some math codes when we switch to another style. As the number of characters that this applies to this is efficient enough for a modern computer. A peek into `luatex-math.tex` gives an impression of what we deal with. However, keep in mind that the implementation in MkIV goes it differently and is therefore more powerful. We also have hardly any definitions at the TEX end and use information from `char-def.lua` instead.

In math mode there is a subtle difference in the way grouping works with styles:

```
text {\scriptstyle script} normal
```

This is the same as:

```
text \bgroup\scriptstyle script\egroup\ normal
```

---

[12] A technical note: in principle the MkIV approach can have a speed penalty compared to a multi–family approach but we don't care too much about it. Also, as we load less fonts the extra overhead gets compensated nicely.

but different from:

```
text \begingroup\scriptstyle script\endgroup\ script
```

This has to do with the fact that a style switch is explicitly registered in the math list and grouping like this is not limiting the scope. In math mode the braced grouping mode actually does create a math group and there the scope of the switch is limited to that group. In practice users will not run into this but they can use macros that use `\begingroup`. Among other reasons, this is why we have a special mathstyle mechanism.

```
\ruledhbox{$x\begingroup\scriptstyle x\endgroup x$} \quad
\ruledhbox{$x\begingroup\setupmathstyle[script]x\endgroup x$} \quad
\ruledhbox{$x{\setupmathstyle[script]x}x$} \quad
\ruledhbox{$x\startmathstyle[script]x\stopmathstyle x$}
```

This gives:

$\boxed{xxx}$ $\boxed{xxx}$ $\boxed{xxx}$ $\boxed{xxx}$

Mechanisms that support the `mathstyle` parameter know how to apply the proper grouping so you don't have to worry there. You can best avoid using the verbose grouping command and stick to braces or the `start-stop` command. An example is the fence mechanism:

```
\definemathfence
  [fancybracket] [bracket]
  [color=darkblue]
\definemathfence
  [smallbracket] [bracket]
  [command=yes,color=darkgreen,mathstyle=small]
\definemathfence
  [normalbracket] [bracket]
  [command=yes,color=darkred]
```

We apply this to an example:

```
$x \fenced[bar]{\frac{1}{x}} x$ \quad
$x \fenced[doublebar]{\frac{1}{x}} x$ \quad
$x \fenced[bracket]{\frac{1}{x}} x$ \quad
$x \fenced[fancybracket]{\frac{1}{x}} x$ \quad
$x \frac{1}{n} \normalbracket{\frac{1}{n}} \smallbracket{\frac{1}{s}} x$
```

Of course these somewhat weird examples are not real but at least they demonstrate the principles.

$$x \left| \frac{1}{x} \right| x \quad x \left\| \frac{1}{x} \right\| x \quad x \left[\frac{1}{x} \right] x \quad x \left[\frac{1}{x} \right] x \quad x \frac{1}{n} \left[\frac{1}{n} \right] \left[\frac{1}{s} \right] x$$

A math style is a combination of the following keys. Their effect can depend on the current state, for instance you can switch cramp or size indepently.

`display`	display style, like text style but somewhat more spacy
`text`	text style, normally used inline
`script`	smaller than text cq. display style
`scriptscript`	smaller than script style
`cramped packed`	more tightly positioned superscripts
`uncramped normal`	normal positioned superscripts
`small`	switch to the next smaller style but keep cramp state
`big`	switch to the next larger style but keep cramp state

Future versions of MkIV will provide more features (like parameter sets driven by keywords). As you might prefer a more symbolic approach we provide:

`\definemathstyle[default][text,cramped]`

After this you can use the keyword `default` which has the advantage that you only need to change one definition in order to get different rendering.

7.6 Supported fonts

As in ConTEXt MKIV I wanted to go ahead with Unicode math as soon as the first version of LuaTEX showed up. Because at that time only Cambria was available I decided to provide virtual Unicode math fonts as a prelude to proper replacements for the popular Type1 math fonts. In the meantime Xits came around and in 2012 we had quite useable math companions for the public Latin Modern, Pagella and Termes fonts and the TEX user groups started shipping OpenType variants of Lucida. The virtual variants will still around so that we can compare them with the new implementations. As the official specification of OpenType math is not always clear from the beginning the OpenType fonts get improved over time. In fact, this is true not only for math fonts. Just think of this:

• As Unicode gets extended, fonts might get more glyphs and possibly alternate shapes.

• The more languages are supported, the more glyphs are to be available and features have to get language dependent instances.

• The larger the font, the bigger the chance that mistakes get unnoticed especially when contextual subtitutions and positioning are used.

• Math fonts can get more script and scriptscript alternates, more size variants, more advanced extensibles, bidirectional support, etc.

So, like regular programs, LuaTEX and macro packages, we now have fonts as component that needs occasional updating. Of course resources like hyphenation patterns are also subjected to this, so it's not a new aspect. But still, best keep en eye on font updates.

While there are lots of text fonts, there are not that many math fonts, so you can safely assume that ConTEXt ships with the proper setup for those fonts. Of course you have to

choose a specific instance when you set up your own combination of fonts, but a peek into the typescripts shows the way.

In the font manual and on the wiki you can find more about typescript and what is possible, so here we just take a look at one definition:

```
\starttypescript [serif] [dejavu] [name]
  \definefontsynonym [Serif]           [name:dejavuserif]           [features=default]
  \definefontsynonym [SerifBold]       [name:dejavuserifbold]       [features=default]
  \definefontsynonym [SerifItalic]     [name:dejavuserifitalic]     [features=default]
  \definefontsynonym [SerifBoldItalic] [name:dejavuserifbolditalic] [features=default]
\stoptypescript

\starttypescript [sans] [dejavu] [name]
  \definefontsynonym [Sans]            [name:dejavusans]            [features=default]
  \definefontsynonym [SansBold]        [name:dejavusansbold]        [features=default]
  \definefontsynonym [SansItalic]      [name:dejavusansoblique]     [features=default]
  \definefontsynonym [SansBoldItalic]  [name:dejavusansboldoblique] [features=default]
\stoptypescript

\starttypescript [mono] [dejavu] [name]
  \definefontsynonym [Mono]            [name:dejavusansmono]            [features=none]
  \definefontsynonym [MonoBold]        [name:dejavusansmonobold]        [features=none]
  \definefontsynonym [MonoItalic]      [name:dejavusansmonooblique]     [features=none]
  \definefontsynonym [MonoBoldItalic]  [name:dejavusansmonoboldoblique] [features=none]
\stoptypescript

\starttypescript[dejavu]
  \definetypeface [dejavu] [rm] [serif] [dejavu] [default]
  \definetypeface [dejavu] [ss] [sans]  [dejavu] [default]
  \definetypeface [dejavu] [tt] [mono]  [dejavu] [default]
  \definetypeface [dejavu] [mm] [math]  [xits]   [default] [scale=1.2]
\stoptypescript
```

So, in many cases you can just copy this blob and replace the font names by your own.

Loading a font, and Dejavu is a predefined one, is done as follows:

```
\setupbodyfont[dejavu]
```

In a similar fashion you can enable `cambria`, `pagella`, `termes`, `lucidaot`, etc. and if you don't use this command at all, you get Latin Modern. These fonts are part of TEX distributions, including ConTEXt stand–alone that can be downloaded from ConTEXt garden.

If you want to use Lucida, all you have to do when you have bought the fonts, is to put the OpenType files in a place where they can be found, for instance:

```
tex/texmf-fonts/fonts/data/lucida
```

Of course you need to run `mtxrun --generate` afterwards so that the files can be found.

Tracing and characters coverage will be discussed here as soon as the styles that are used for them are normalized.

7.7 Stylistic alternates

Some fonts provide stylistic alternates. These can be described in goodies files and the Lucida setup is a good example. Here we demonstrate the effects. We disable the default math rendering (which takes the italic variants).

```
\switchtobodyfont[lucidaot,14.4pt]
\setupmathrendering[lucidaot][it=]
$x
   ^{i \leftarrow 0 = ø}
   _{i \leftarrow 0 = ø}
$
```

The next code enabled three alternatives:

```
\switchtobodyfont[lucidaot,14.4pt]
\setupmathrendering[lucidaot][it=]
$x
   ^{i \leftarrow 0 = ø}
   _{\setmathfontalternate{arrow}
     \setmathfontalternate{dotless}
     \setmathfontalternate{zero}
     i \leftarrow 0 = ø}
$
```

Here we set them in one go:

```
\switchtobodyfont[lucidaot,14.4pt]
\setupmathrendering[lucidaot][it=]
$x
   ^{i \leftarrow 0 = ø}
   _{\setmathfontalternate{arrow,dotless,zero}
     i \leftarrow 0 = ø}
$
```

The last example shows how to enable these features globally:

```
\switchtobodyfont[lucidaot,14.4pt]
```

Math

```
\setupmathrendering[lucidaot][it=]
\setupmathematics[stylealternative={arrow,dotless,zero}]
$x
   ^{i \leftarrow 0 = ø}
   _{i \leftarrow 0 = ø}
$
```

The results are collected here:

$$x_{i\leftarrow 0=\varnothing}^{i\leftarrow 0=\varnothing} \qquad x_{\imath\leftarrow 0=\emptyset}^{i\leftarrow 0=\varnothing} \qquad x_{\imath\leftarrow 0=\emptyset}^{i\leftarrow 0=\varnothing} \qquad x_{\imath\leftarrow 0=\emptyset}^{\imath\leftarrow 0=\emptyset}$$

nothing   **stepwise**   **combined**   **global**

7.8 Italics and limits

An OpenType font treats italic correction differently from traditional fonts. Officially the italic correction is used for placement above and below limits where the scripts shift left and right half of the correction from the center of the shape. Advanced kerns are then to be used for anchoring the scripts when they are placed at the right side (so far no fonts seem to do this). Because we cannot foresee if fonts compensate for correction then we can control placement a bit. There is a parameter \mathnolimitsmode that controls the correction.
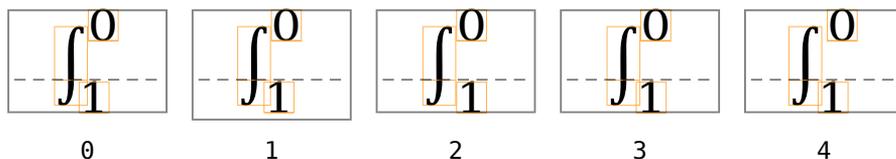
A value larger than 15 is interpreted as a factor (in the usual TeX way 1000 means 1.0). We have some values left for future use when correction is to be combined with kerns.

In ConTeXt we set the value to 1 which means that the factors for super- and subscript are set via math parameters (or constants in the font). We use a default of {0,800} so we don't shift the superscript and the subscript we shift less than the italic correction. This is driven by a feature but you can change the values before loading a font, for instance with:

```
\adaptfontfeature[*math*][mathnolimitsmode={100,700}]
```

The defaults come out as:
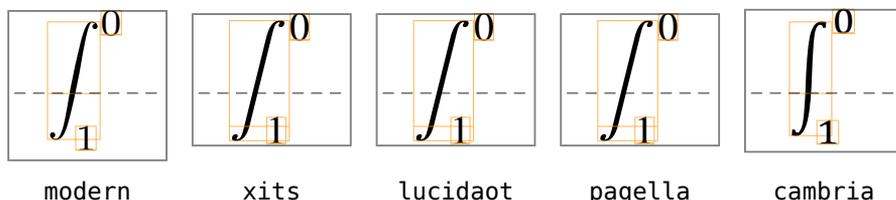
8 Extensions

8.1 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.

8.2 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.

An italic correction is bound to a glyph and bound to a font. In figure 8.1 we see how an italic shape extends out of the bounding box. This is not the case in Dejavu: watch figure 8.2.

Latin Modern
Roman Regular

Latin Modern
Roman Italic

Figure 8.1   Italic overshoot in Latin Modern.

Dejavu Regular   Dejavu Italic

Figure 8.2   Italic overshoot in Dejavu Serif.

This means that the application of italic correction should never been applied without knowing the font. In figure 8.3 we see an upright word following an italic. The space is determined by the upright one.

Latin Modern   Dejavu

Figure 8.3   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 8.4 we see italic correction in action.

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


Figure 8.4   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 8.5. 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 enables). In the given example we used:

```
\setupitaliccorrection
  [text]
```

You can combine keys:

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


Figure 8.5 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. These 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]
```

8.3 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.

8.4 Slanting

This 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.

8.5 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.

8.6 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}]
```

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 8.6 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).

| default | 1em 1ex 1ex | 1em 2ex 3ex | 1em 3ex 2ex | 3em 3ex 3ex |
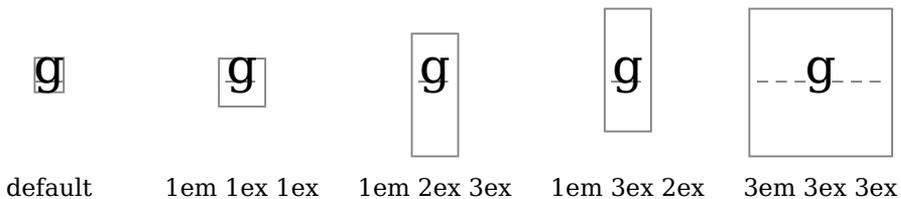
Figure 8.6   Freezing dimensions of glyphs.

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:

clipped clipped clipped

8.7 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` upto `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. We already discussed this in section 5.4 so we only repeat the usage.

```
\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.

8.8 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.[13]

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		

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
```

---

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

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+000A1	0.00	¡	0.20	U+02018	0.70	'	0.70
U+00028	0.05	(	0.00	U+000AB	0.50	«	0.50	U+02019	0.00	'	0.70
U+00029	0.00	)	0.05	U+000AD	0.00		0.70	U+0201A	0.70	‚	0.00
U+0002C	0.00	,	0.70	U+000BB	0.50	»	0.50	U+0201B	0.70	‛	0.00
U+0002D	0.00	-	0.70	U+000BF	0.00	¿	0.20	U+0201C	0.50	"	0.50
U+0002E	0.00	.	0.70	U+0060C	0.00		0.70	U+0201D	0.00	"	0.50
U+0003A	0.00	:	0.50	U+0061B	0.00		0.50	U+0201E	0.50	„	0.00
U+0003B	0.00	;	0.50	U+0061F	0.00		0.20	U+0201F	0.50	‟	0.00
U+0003F	0.00	?	0.20	U+006D4	0.00		0.70	U+02039	0.70	‹	0.70
U+0005B	0.05	[	0.00	U+02013	0.00	–	0.30	U+0203A	0.70	›	0.70
U+0005D	0.00	]	0.05	U+02014	0.00	—	0.20				

protrusion vector alpha

U+00041	0.05	A	0.05	U+0004B	0.00	K	0.05	U+00056	0.05	V	0.05
U+00046	0.00	F	0.05	U+0004C	0.00	L	0.05	U+00057	0.05	W	0.05
U+0004A	0.05	J	0.00	U+00054	0.05	T	0.05	U+00058	0.05	X	0.05

```
U+00059  0.05  Y  0.05     U+00074  0.00  t  0.05     U+00078  0.05  x  0.05
U+0006B  0.00  k  0.05     U+00076  0.05  v  0.05     U+00079  0.05  y  0.05
U+00072  0.00  r  0.05     U+00077  0.05  w  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 use the following example. The results are shown in figure 8.7. The colored text is the protruding one.

```
\startoverlay
    {\ruledvbox \bgroup
        \hsize\textwidth
        \MyTestA
        \setupalign[normal]
```

Extensions

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

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 8.7   The difference between no protrusion and quality protrusion.

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 8.8 we see that the somewhat extreem 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 8.8   The influence of extreme protrusion on preceding characters.

8.9 Expansion

Expansion is also an inheritance of pdfTeX.[14] This mechanism selectively expands characters, normally upto 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 wrong.

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 then 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's best not to use too many stretch and shrink steps. A stretch of 2 and shrink of 2 and step of .25 results in upto 8 instances plus the regular sized one.

expansion vectors

We only have one vector: `quality`:

U+00032	2	0.70	U+0004D	M	0.70	U+00065	e	0.70
U+00033	3	0.70	U+0004E	N	0.70	U+00067	g	0.70
U+00036	6	0.70	U+0004F	O	0.50	U+00068	h	0.70
U+00038	8	0.70	U+00050	P	0.70	U+0006B	k	0.70
U+00039	9	0.70	U+00051	Q	0.50	U+0006D	m	0.70
U+00041	A	0.50	U+00052	R	0.70	U+0006E	n	0.70
U+00042	B	0.70	U+00053	S	0.70	U+0006F	o	0.70
U+00043	C	0.70	U+00055	U	0.70	U+00070	p	0.70
U+00044	D	0.50	U+00057	W	0.70	U+00071	q	0.70
U+00045	E	0.70	U+0005A	Z	0.70	U+00073	s	0.70
U+00046	F	0.70	U+00061	a	0.70	U+00075	u	0.70
U+00047	G	0.50	U+00062	b	0.70	U+00077	w	0.70
U+00048	H	0.70	U+00063	c	0.70	U+0007A	z	0.70
U+0004B	K	0.70	U+00064	d	0.70			

---

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

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 8.9. 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
```

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 8.9   The difference between no expansion and quality 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:

negative   positive

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

negative   positive

The reason that we mention this is that when we apply OpenType features, positioning not necessarily result 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 "real 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 8.10 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 8.11. 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]
```


no hz & hz

no hz & full hz

hz & full hz

Figure 8.10   The two expansion methods compared.

extreme: no hz & hz

extreme: no
hz & full hz

extreme: hz
& full hz

regular: no hz & hz

regular: no
hz & full hz

regular: hz & full hz

minimal: no hz & hz

minimal: no
hz & full hz

minimal: hz
& full hz
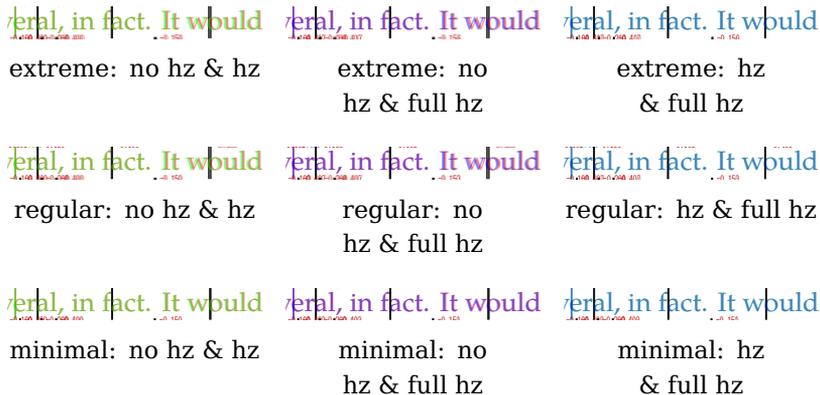
Figure 8.11   The two expansion methods compared (zoomed in).

In ConTEXt the hz alignment option only enables expansion of glyphs, while fullhz also applies it to kerns. However, in the examples here we had to explicitly enable font kerns in node mode:

```
\enabledirectives[fonts.injections.fontkern]
```

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.

8.10 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 > V̋ (U+00476) = V (U+00474) + ̋ (U+0030F)
fonts > combining > v̋ (U+00477) = v (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 real 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]
```


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,
      },
    },
```

Extensions

```
  },
}
```

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.

8.11 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 careful.

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	0.00	rubish	0.00
	how to violate a proper font design		rubish	

Extensions

0.005	how to violate a proper font design	0.83	rubish	0.77
	how to violate a proper font design		rubish	
0.010	how to violate a proper font design	1.64	rubish	1.52
	how to violate a proper font design		rubish	
0.015	how to violate a proper font design	2.44	rubish	2.26
	how to violate a proper font design		rubish	
0.020	how to violate a proper font design	3.22	rubish	2.99
	how to violate a proper font design		rubish	
0.025	how to violate a proper font design	3.99	rubish	3.72
	how to violate a proper font design		rubish	
0.030	how to violate a proper font design	4.76	rubish	4.43
	how to violate a proper font design		rubish	
0.035	how to violate a proper font design	5.50	rubish	5.13
	how to violate a proper font design		rubish	
0.040	how to violate a proper font design	6.24	rubish	5.82
	how to violate a proper font design		rubish	
0.045	how to violate a proper font design	6.97	rubish	6.49
	how to violate a proper font design		rubish	

8.12 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.

8.13 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 / fitzen / effe fietsen / ch ck ft tz fi 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 / fitzen / effe fietsen / ch cf ft tz fi fi
liga + keepligatures                  sitzen / fitzen / effe fietsen / ch cf ft tz fi fi
liga + ccmp                           sitzen / fitzen / effe fietsen / ch cf ft tz fi fi
liga + ccmp + keepligatures           sitzen / fitzen / effe fietsen / ch cf ft tz fi fi
ccmp + keepligatures                  sitzen / fitzen / effe fietsen / ch cf ft tz fi 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 / f i t z e n / e f f e f i e t s e n / c h c f f t t z f i f i
liga + keepligatures                  s i t z e n / f i t z e n / e f f e f i e t s e n / c h c f f t t z f i f i
liga + ccmp                           s i t z e n / f i t z e n / e f f e f i e t s e n / c h c f f t t z f i f i
liga + ccmp + keepligatures           s i t z e n / f i t z e n / e f f e f i e t s e n / c h c f f t t z f i f i
ccmp + keepligatures                  s i tz e n / fi tz e n / e f f e f i e t s e n / ch cf ft tz fi fi
```

In the next example we use \letterspacing:

```
liga                                  s i tz e n / fi tz e n / e f f e f i e t s e n / ch cf ft tz fi fi
liga + keepligatures                  s i tz e n / fi tz e n / e f f e f i e t s e n / ch cf ft tz fi fi
liga + ccmp                           s i tz e n / fi tz e n / e f f e f i e t s e n / ch cf ft tz fi fi
liga + ccmp + keepligatures           s i tz e n / fi tz e n / e f f e f i e t s e n / ch cf ft tz fi fi
ccmp + keepligatures                  s i tz e n / fi tz e n / e f f e f i e t s e n / ch cf ft tz fi fi
```

The difference is that the letterspacing variant dynamically adds the predefined feature-set `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, -- ft
  },
}
```

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	PEL·LÍCULES	PEL·LÍCULES
pel·lícules	pel·lícules	pel·lícules	pel·lícules	pel·lícules
PEL·LÍCULES	PEL·LÍCULES	PEL·LÍCULES	PEL·LÍCULES	PEL·LÍCULES
pel·lícules	pel·lícules	pel·lícules	pel·lícules	pel·lícules

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.

8.14 New features

8.14.1 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
```

8.14.2 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.

8.14.3 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`	abracadabra
multiple	`mtest`	abracadabra
ligature	`ltest`	abracadabra
kern	`ktest`	abracadabra
pair	`ptest`	abracadabra
chain sub	`ctest`	abracadabra

8.14.4 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.

8.14.5 Language dependencies

When OpenType was not around we only had to deal with ligatures, smallcaps and old-

style 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" },
    },
}
\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[Serif*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 diearesis has a different meaning.

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

The above is 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[Serif*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[Serif*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.

Extensions

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[Serif*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[Serif*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[Serif*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.

8.14.6 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>, ... },
                    },
                },
            },
        },
    },
}
```

8.14.7 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:

av

ery

long

word

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]
```

8.14.8 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.

8.15 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:

test test test test test	test test test test test
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 un 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 dreived 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 valua 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 an `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.

8.16 Collections

Todo.

9 Hooks

9.1 Introduction

One of the virtues of TEX is its flexibility. Because we cannot predict what users want to mess around with, much of the underlying code has hooks. And because it's not too hard to add functionality that will break things we will not advocate all of it. Of course you can study the code and figure out what can be done and there is no problem with that. It's just that you shouldn't expect much support.

In this chapter we collect some of these hooks. If you run into interesting ones that are worth mentioning, you can always ask us to add description here.

9.2 Safe hooks

9.2.1 Trimming fonts

Because we store font related information in Lua tables there can be situations where the resources used outgrow memory. An example of such a font is lastresort that basically defined the whole Unicode range. The font is actually not that large as it uses similar placeholders for glyphs in a range, but it has rather verbose (redundant) names. As we normally don't need these, you can decide to strip them away.

```
\startluacode
    fonts.handlers.otf.readers.registerextender {
        name  = "remove names from lastresort",
        action = function(fontdata)
            if fontdata.metadata.fullname == "LastResort" then
                for k, v in next, fontdata.descriptions do
                    v.name = nil
                end
            end
        end
    }
\stopluacode

\definedfont[LastResort][lastresort*default sa 1]
```

This will result in a much smaller font, one that has less change to crash the engine due to lack of memory. Extenders like this are applied once the font has been loaded but before it gets saved.

9.3 Loading

9.3.1 Introduction

We basically have to deal with three font formats that can easily be recognized by the suffix of the files involved: `tfm` and `vf` files that describe 8 bit fonts, traditionally bitmap fonts, but as they carry only metric information, any 8 bit font can be described. Then there are `afm` files that contain metrics related to Type1 fonts (stored in `pfb` files). Although such fonts could contain more than 256 shapes, the implementation was limited to 8 bits too. By converting `afm` files to `tfm` files, traditional TeX can deal with Type1 given that the backend can include them in the final result.

In this section we will discuss some aspects of the OpenType font reader. As TeX only deals with metrics (in the frontend) we need to parse them, filter information from it and pass the metrics to TeX. In addition, we can use all kind of extra information to manipulate the so called node list but in the end TeX is only interested in font id's (that point to a font resource) and glyph indexes.

To overcome the 256 limitation of Type1 fonts, in ConTeXt we moved away from `tfm` files (we can of course still deal with them) and turn `afm` files into so called wide fonts. Basically we turn them in a more rich format that looks similar to the internal OpenType format we use. We will not go into much detail about that because Type1 is kind of obsolete and being replaced by OpenType, but we will of course support the old formats simply because we have all these fonts around.

Already early in the development of LuaTeX a font loader library was created that can turn an OpenType (but also a Type1) font into a Lua table. This library is derived from FontForge which makes it possible to look into a font using that editor and at the same time get a similar view on the font in Lua, which is quite handy. However, at some point in ConTeXt we wanted to play with outlines in MetaPost and for that purpose an OpenType reader was written in Lua that could extract the data. Because Type1 fonts already were done in Lua it was a logical step to also do OpenType in Lua so now we use an alternative loader that doesn't depend in the FontForge library. This not only gives more flexibility but also makes it possible to avoid some conversions needed to provide the ConTeXt font handler with the needed information in an efficient way.

9.3.2 Loading OpenType fonts

As with most binary media formats today an OpenType font file is a linked list of records. The top level structure is called table. There are two flavours of OpenType where the main difference is in the way the shapes are defined: they can be TrueType outlines using quadratric bezier curves or cff files using cubic bezier curves. The last variant is the same as PostScript Type1 fonts. Simplified, a quadratic curve defines the shape in points with a control point in between, while a quadratic one also has points but each with two control points (as in MetaPost).

An OpenType font can be large: there can be upto 65536 glyphs and lots of extra properties and features. In order to save space the data is rather packed using different numeric data types. Of course one can wonder if size really matters now that most bandwidth is taken by audio, video and pictures but we have to live with it.

The definition of OpenType can be found on the Microsoft website: https://www.microsoft .com/typography/otspec. Most tables then could make sense for us are mentioned in the following list:

required	cmap	character to glyph mapping
	head	font header
	hhea	horizontal header
	hmtx	horizontal metrics
	maxp	maximum profile
	name	naming table
	os/2	os/2 and windows specific metrics
	post	postScript information
truetype	glyf	glyph data
	loca	index to location
postscript	cff	compact font format
	vorg	vertical origin
typographic	base	baseline data
	gdef	glyph definition data
	gpos	glyph positioning data
	gsub	glyph substitution data
	jstf	justification data
	math	math layout data
extras	kern	kerning
	ltsh	linear threshold data
	vhea	vertical metrics header
	vmtx	vertical metrics
	colr	color table
	cpal	color palette table

When we read these tables it depends on what we want to do with the result how much we will really read. For instance when we only want to identify a font and get some basic information we don't need to read all tables and certainly don't need to read them completely. If we want to have the outlines we need to read the `glyf` or `cff` table. If we also want to boundingbox of PostScript shapes we even need to process the shapes so that we know the dimensions of the result. There is no need to summarize the format here in detail because you can find it on the Microsoft site. Here I only cover some aspects that influence the way TeX can use the fonts.

One of the main differences between the readers is that the FontForge reader has a lot of (recovery) heuristics for bad fonts. Nowadays most fonts are quite okay, and in ConTeXt we prefer to just reject bad ones. In the process of loading the built-in loader gives

each glyph a name (it makes them up for variants needed for features). It also tries to figure out some font properties, like the weight. If does a pretty good job on that but it is also hard to repair at the Lua end when it makes a bad guess. The Lua variants stays closer to the specification, but delegates more to the final user, which is good because we need and want that level of control as controls is what TEX is about. It also made it possible to support for instance colored fonts without too much effort.

So what data needs to be collected? If we look at what we get eventually the list of glyphs is the bulk. For each glyph we collect some metric information. For instance we fetch the (advance) width of the glyph but also the boundingbox, which gives us the the height and depth.

In the font file the list of glyphs starts at zero and runs up tot the total number of glyphs. The index in this table is used in for instance the tables that define the font features, for instance kerning between glyphs, or multiple glyphs that are turned into ligatures. Each glyph gets a name. That can be a meaningful one but also a rather dumb one, for instance the index number.

Eventually (at least in ConTEXt) we don't order by glyph index but by Unicode. The font file contains information about the mapping from index to Unicode. In principle other encodings are possible but we stick to Unicode. But, because many glyphs can refer to one Unicode slot, for instance a regular shape as well as a smallcaps or oldstyle variant. These extra glyphs we let end up in the private Unicode areas. This also means that with each glyph in the final table there is also a field that has the Unicode. Because we order by Unicode we also need to store the index. An example from a Latin Modern font is:

```
[97] = {
    boundingbox = { 34, -10, 474, 446 },
    index       = 28,
    name        = "a",
    unicode     = 97,
    width       = 490,
}
```

Another example is the following. Here we end up in private space:

```
[983059] = {
    boundingbox = { 30, -10, 734, 446 },
    index       = 19,
    name        = "oe.dup",
    unicode     = 339,
    width       = 762,
}
```

Yet another entry is:

```
[306] = {
    boundingbox = { 28, -22, 790, 683 },
    index       = 357,
    name        = "I_J",
    unicode     = { 73, 74 },
    width       = 839,
  },
```

Here you see two Unicode numbers. That kind of information is deduced from the name of the glyph, using knowledge on how such names are supposed to be constructed, or, when that is not possible, from ligature information in the fonts.

It makes no sense to discuss the whole font table in detail, if only because most users will never (need to) see it. But if your curious you can have a look at the fonts in the cache tree, in the ConTEXt distribution from the ConTEXt garden this is

```
.../tex/texmf-cache/luatex-cache/context/<somehash>/fonts/otl
```

There can be three kind of files there, with suffixes `tma`, `tmc` and `tmb`. The first one is the table as converted from the binary font file. The second and third variants are just bytecode compilations of this file (for LuaTEX and/or LuajitTEX). The bytecode variants are smaller but more important, they load a bit faster. On my disk the largest `tma` file is just below 10 MByte (an extensive cjk font) but normally they are in the few hundred KByte range (some are real small), with the bytecode files of course being relatively small to their original.

However, there is a bit of cheating here. If we run the command:

```
mtxrun --script font --convert lmroman10-regular.otf
```

A Lua file is generated: `lmroman10-regular.lua`. This file is much larger than the `tma` file in the cache:

```
643.924  lmroman10-regular.lua0.029
209.950  lmroman10-regular.tma0.010
121.541  lmroman10-regular.tmb
134.564  lmroman10-regular.tmc0.003
```

The reason for this is the following. Most information is stored in tables. Especially tables that describe font features can be the same all over the place. This is why we pack the table in a more compact format before saving it in the cache, and unpack it after loading. The effects on loading are neglectable but and it has the benefit that it saves a lot of memory. By looking at such numbers one should be careful with conclusions, but (assuming proper garbage collection) we see a memory footprint of the `lua` file of 2836 Kbyte, while the unpacked variant takes 704 Kbyte. You can imagine what happens with large cjk fonts. Loading the (larger unpacked) `lua` file currently costs me 0.029 seconds, while loading and unpacking the `tma` file takes 0.010 seconds and the bytecode variant `tmc` 0.003 seconds.

9.3.3 Loading Type1 fonts

When we started with ConTEXt MkIV (which is shortly after we started with LuaTEX) the only tfm files that were loaded, were those to make virtual Unicode math fonts, awaiting real OpenType math fonts. Math fonts are kind of special with respect to metrics and such.

For Type1 text fonts we didn't use the tfm files but went for parsing afm files. That way we could use all the glyphs provided by fonts and not be limited to 256 slots. So, effectively we made them Unicode and similar to OpenType. Of course the only features were ligatures, kerns and some special ones like TEX ligatures and replacements. With the old loader code, we always made them base mode fonts, which means that processing was delegated to TEX. In the new loader we implement ligatures and kerns as node mode features, which means that we can use those fonts in base mode as well as node mode. The last options therefore permits to add or adapt features to Type1 fonts as well.

In the next sections we will focus on OpenType but as the Type1 fonts are organized in a similar way, some of it also applies to this older type. The most important to keep in mind is that we only have `liga`, `kern` and a few ConTEXt specific features.

9.4 The tables

9.4.1 Structure

Getting a font read for TEX happens in stages. The original OpenType file is read only once. At that moment the shapes are described in the `descriptions` subtable while by the time that we pass the information to TEX they are in `characters`. The reason is that we go from dimensions in font units to dimensions in scaled points. We start with the following table:

```
t={
 ["cache_uuid"]="<string>",
 ["cache_version"]="<float>",
 ["compacted"]="<boolean>",
 ["creator"]="<string>",
 ["descriptions"]={
  {
   ["boundingbox"]={ "<units>", "<units>", "<units>", "<units>" },
   ["depth"]="<units>",
   ["height"]="<units>",
   ["index"]="<index>",
   ["italic"]="<units>",
   ["math"]={
    ["accent"]="<units>",
    ["hparts"]={
```

```
 {
  ["advance"]="<scaled>",
  ["end"]="<scaled>",
  ["extender"]="<scaled>",
  ["glyph"]="<unicode>",
  ["start"]="<scaled>",
 },
},
["hvariants"]={ "<array>" },
["kerns"]={
 ["bottomleft"]={
  {
   ["height"]="<scaled>",
   ["kern"]="<scaled>",
  },
 },
 ["bottomright"]={
  {
   ["height"]="<scaled>",
   ["kern"]="<scaled>",
  },
 },
 ["topleft"]={
  {
   ["height"]="<scaled>",
   ["kern"]="<scaled>",
  },
 },
 ["topright"]={
  {
   ["height"]="<scaled>",
   ["kern"]="<scaled>",
  },
 },
},
["vparts"]={
 {
  ["advance"]="<scaled>",
  ["end"]="<scaled>",
  ["extender"]="<scaled>",
  ["glyph"]="<unicode>",
  ["start"]="<scaled>",
 },
},
```

```
      ["vvariants"]={ "<array>" },
     },
     ["unicode"]="<unispec>",
     ["width"]="<units>",
    },
   },
   ["format"]="<string>",
   ["goodies"]="<hash>",
   ["metadata"]={
    ["ascender"]="<units>",
    ["averagewidth"]="<units>",
    ["capheight"]="<units>",
    ["descender"]="<units>",
    ["family"]="<string>",
    ["familyname"]="<string>",
    ["fontname"]="<string>",
    ["fullname"]="<string>",
    ["italicangle"]="<float>",
    ["monospaced"]="<boolean>",
    ["panoseweight"]="<string>",
    ["panosewidth"]="<string>",
    ["pfmweight"]="<units>",
    ["pfmwidth"]="<units>",
    ["subfamily"]="<string>",
    ["subfamilyname"]="<string>",
    ["subfontindex"]="<index>",
    ["units"]="<cardinal>",
    ["version"]="<string>",
    ["weight"]="<string>",
    ["width"]="<string>",
    ["xheight"]="<units>",
   },
   ["private"]="<unicode>",
   ["properties"]={
    ["hascolor"]="<boolean>",
    ["hasitalics"]="<boolean>",
    ["hasspacekerns"]="<boolean>",
   },
   ["resources"]={
    ["duplicates"]="<hash>",
    ["features"]={
     ["gpos"]="<hash>",
     ["gsub"]="<hash>",
    },
```

Hooks

```
  ["filename"]="<string>",
  ["markclasses"]="<hash>",
  ["marks"]="<hash>",
  ["marksets"]="<hash>",
  ["mathconstants"]="<hash>",
  ["private"]="<cardinal>",
  ["sequences"]="<array>",
  ["version"]="<string>",
 },
 ["size"]="<cardinal>",
 ["tableversion"]="<float>",
 ["time"]="<cardinal>",
}
```

The table passed TeX is constructed from this one and looks like:

```
t={
 ["characters"]={
  {
   ["accent"]="<scaled>",
   ["commands"]={
    { "<keyword>", "<value>" },
   },
   ["depth"]="<scaled>",
   ["expansion_factor"]="<scaled>",
   ["height"]="<scaled>",
   ["hvariants"]={ "<array>" },
   ["index"]="<index>",
   ["italic"]="<scaled>",
   ["kerns"]="<hash>",
   ["left_protruding"]="<scaled>",
   ["ligatures"]="<hash>",
   ["next"]="<array>",
   ["right_protruding"]="<scaled>",
   ["tounicode"]="<string>",
   ["unicode"]="<unispec>",
   ["vvariants"]={ "<array>" },
   ["width"]="<scaled>",
  },
 },
 ["descriptions"]={
  {
   ["boundingbox"]={ "<units>", "<units>", "<units>", "<units>" },
   ["depth"]="<units>",
   ["height"]="<units>",
```

```
["index"]="<index>",
["italic"]="<units>",
["math"]={
 ["accent"]="<units>",
 ["hparts"]={
  {
   ["advance"]="<scaled>",
   ["end"]="<scaled>",
   ["extender"]="<scaled>",
   ["glyph"]="<unicode>",
   ["start"]="<scaled>",
  },
 },
 ["hvariants"]={ "<array>" },
 ["kerns"]={
  ["bottomleft"]={
   {
    ["height"]="<scaled>",
    ["kern"]="<scaled>",
   },
  },
  ["bottomright"]={
   {
    ["height"]="<scaled>",
    ["kern"]="<scaled>",
   },
  },
  ["topleft"]={
   {
    ["height"]="<scaled>",
    ["kern"]="<scaled>",
   },
  },
  ["topright"]={
   {
    ["height"]="<scaled>",
    ["kern"]="<scaled>",
   },
  },
 },
 ["vparts"]={
  {
   ["advance"]="<scaled>",
   ["end"]="<scaled>",
```

Hooks

```
      ["extender"]="<scaled>",
      ["glyph"]="<unicode>",
      ["start"]="<scaled>",
     },
    },
    ["vvariants"]={ "<array>" },
  },
  ["unicode"]="<unispec>",
  ["width"]="<units>",
 },
},
["parameters"]={
 ["ascender"]="<scaled>",
 ["descender"]="<scaled>",
 ["designsize"]="<scaled>",
 ["expansion"]={
  ["auto"]="<boolean>",
  ["shrink"]="<scale>",
  ["step"]="<scale>",
  ["stretch"]="<scale>",
 },
 ["extendfactor"]="<float>",
 ["factor"]="<float>",
 ["hfactor"]="<float>",
 ["mathsize"]="<cardinal>",
 ["protrusion"]={
  ["auto"]="<boolean>",
 },
 ["quad"]="<scaled>",
 ["scaledpoints"]="<scaled>",
 ["scriptpercentage"]="<float>",
 ["scriptscriptpercentage"]="<float>",
 ["size"]="<scaled>",
 ["slantfactor"]="<float>",
 ["slantperpoint"]="<scaled>",
 ["spacing"]={
  ["extra"]="<scaled>",
  ["shrink"]="<scaled>",
  ["stretch"]="<scaled>",
  ["width"]="<scaled>",
 },
 ["units"]="<scaled>",
 ["vfactor"]="<float>",
 ["xheight"]="<scaled>",
```

```
 },
 ["properties"]={
  ["autoitalicamount"]="<float>",
  ["cidinfo"]="<hash>",
  ["embedding"]="<cardinal>",
  ["encodingbytes"]="<cardinal>",
  ["filename"]="<string>",
  ["finalized"]="<boolean>",
  ["fontname"]="<string>",
  ["format"]="<string>",
  ["fullname"]="<string>",
  ["hasitalics"]="<boolean>",
  ["hasmath"]="<boolean>",
  ["mathitalics"]="<boolean>",
  ["mode"]="<string>",
  ["name"]="<string>",
  ["noglyphnames"]="<boolean>",
  ["nostackmath"]="<boolean>",
  ["psname"]="<string>",
  ["textitalics"]="<boolean>",
  ["virtualized"]="<boolean>",
 },
}
```

There might be a few more (often obscure) fields for special purposes. The characters subtable conforms to what TeX expects, while the descriptions stays closer to OpenType. The `kerns` and `ligatures` subtables are there for base mode and are not present in `node` mode. The `commands` and `fonts` subtables relate to virtual fonts.

- Start with the (already) loaded OpenType table.
- Copy relevant information from `descriptions` to `characters` etc.
- Construct `properties` and `parameters` tables.
- Apply additional manipulators, for instance extend the `characters` table, with expansion and protrusion.
- Scale the `characters`, `properties` and `parameters`.
- Apply additional manipulators.
- Pass the table to TeX, but keep it around for later access.

One of the things you need to be aware of is that all references to glyphs are Unicode slots, either natural ones (representing a character) or a private one (representing an alternative representation). In OpenType features are defined in terms of glyph indices but we prefer Unicode because that is easier to deal with when we run over the node list. Before font processing the character field in a glyph node is a Unicode slot and afterwards it's still a Unicode but when it's a private one it can always be resolved to a non private slot of sequence of slots. Of course that could also be done with indices but it's just more natural this way.

Hooks

Another thing to note is that in the descriptions we're still working with font units ranging from −1000 to +1000, −2048 to +2048 or similar ranges. At the TEX end we need scaled points which are much larger numbers.

The question is: how often do users need to access the raw data in a font? After a decade of MkIV and LuaTEX hardly any user has requested such access, probably because when needed easier interfaces were provided. Also, in the ConTEXt distrubution there are some examples of manipulations that can be copied and adapted to personal use. There's also a danger is messing with the fonts (similar messing with the node lists): you never know how it interferes with other (maybe future) features.

If you still want to do it, best is probably to start with saving the to-be-passed-to-TEX table in a file and have a look at it. The most prominent subtable is the `characters` table and messing a bit with dimensions is rather harmless. You could add characters, for instance virtual ones, which again is harmless unless you use invalid commands. You probably want to stay away from the resources subtable, if only because some of its subtables are shared and therefore adapting them can have side effects. The top level `shared` and `unscaled` subtable are off limits as is the `specification`.

You can save a font by consulting one of the hashes but for a specific font you need to know its id. You can do this by using low level accessors but better is to use the helpers made for this, because they prevent saving redundant data.

```
\startluacode
fonts.tables.save  {
    filename = "temp-font-scaled.lua",
    fontname = "dejavusansmono*default",
    method   = "original",
}
\stopluacode
```

At the TEX end you can use:

```
\savefont
  [name=dejavusansmono*default,
   file=temp-o.lua,
   method=original]
\savefont
  [name=dejavusansmono*default,
   file=temp-s.lua,
   method=scaled]
```

When no `name` is given, the current font is used and when no `file` is given a filename is made up. The default `method` is `scaled`. The saved name is reported.

9.4.2 Plug-ins

There are several places where you can hook in code: before scaling (initalizers), after scaling (manipulators) and while processing (processors). Only the first two are meant for tweaks.

```
local do_something = {
    name        = "something",
    description = "doing something",
    initializers = {
     -- position = 1,
        base     = function(tfmdata,value,features) ... end,
        node     = function(tfmdata,value,features) ... end,
    },
    manipulators = {
     -- position = 1,
        base     = function(tfmdata,feature,value) ... end,
        node     = function(tfmdata,feature,value) ... end,
    },
    processors = {
     -- position = 1,
        base     = function(tfmdata,font,attr) ... end,
        node     = function(tfmdata,font,attr) ... end,
    }
}

fonts.constructors.features.register.otf(so_something)
fonts.constructors.features.register.afm(so_something)
```

A `initializer` is applied just before the font gets scaled. This means that the characterm properties and parameters are unscaled! Initializers can for instance be used to add extra features to fonts. You can provide an `position` key with a number to force a place in the list of initializers but of course you can never be sure of interference.

A `manipulator` is applied when the font is scaled but before it gets passed to TeX. It's a good place to tweak dimensions. Here you can also probide a `position`.

The processors are applied when the node list gets processed, hence the `font` and optional `attr` arguments. The action is only applied to the specified font (id) and when an attribute gets passed, this is tested for a value. When an attribute is used, an unset attribute on the node will skip the action.

If adapting characters and their properties is your main objetive, then there is a better plugin mechanism using sequencers. We illustrate this with a fake example:

```
\startluacode
```

```
function document.b_copying(tfmdata)
    logs.report("fonts","before copying: %s",tfmdata.properties.filename)
end
function document.a_copying(tfmdata)
    logs.report("fonts","after copying: %s",tfmdata.properties.filename)
end

function document.b_math(tfmdata)
    logs.report("fonts","before math: %s",tfmdata.properties.filename)
end
function document.a_math(tfmdata)
    logs.report("fonts","after math: %s",tfmdata.properties.filename)
end

utilities.sequencers.appendaction(
    "beforecopyingcharacters",
    "before",
    "document.a_copying"
)

utilities.sequencers.appendaction(
    "aftercopyingcharacters",
    "after",
    "document.b_copying"
)

utilities.sequencers.appendaction(
    "mathparameters",
    "before",
    "document.b_math"
)

utilities.sequencers.appendaction(
    "mathparameters",
    "after",
    "document.a_math"
)
\stopluacode
```

When we call the next command:

```
\definedfont[MathRoman at 3pt]
```

we get this reported:

```
fonts > before math: ...../public/dejavu/texgyredejavu-math.otf
```

```
fonts > after math: ...../public/dejavu/texgyredejavu-math.otf
fonts > after copying: ...../public/dejavu/texgyredejavu-math.otf
fonts > before copying: ...../public/dejavu/texgyredejavu-math.otf
```

In between `before` and `after` we have `system` which is reserved for ConTEXt actions. These actions are executed in the scaler function. The function get two tables passed: the original data as well as the target. If you ever need these hooks, you can probably best run an `inspect` on these arguments to see what you're dealing with.

Fonts get reused when possible and for that a hash is calculated depending on the enabled features and size. If for some reason you want to adapt that hash you can use postprocessors. When the `tfmdata` table has a subtable `postprocessors`, then the actions in that subtable will be applied. When an action returns a string, the string will be combined with the hash. You can set (o rextend) the postprocessors table using the previopusly mentioned commands. However, in ConTEXt you can best stay away from this as it might interfere. This mechanism is mostly provided for generic use.

9.5 Goodies

The font goodies are already discussed as an official mechanism to extend or enhance fonts with additional features. There are quite some goodies defined and for sure more will show up. Here is the full repertoire:

```
t={
 ["author"]="<string>",
 ["colorschemes"]={
  ["default"]={
   { "<string>" },
  },
 },
 ["comment"]="<string>",
 ["compositions"]={
  ["<string>"]={
   ["<unicode>"]={
    ["anchors"]={
     ["bottom"]={},
     ["top"]={},
    },
   },
  },
 },
 ["copyright"]="<string>",
 ["designsizes"]={
  ["<string>"]={
   ["<string>"]="<string>",
```

```
    ["default"]="<string>",
   },
  },
  ["featuresets"]={
   ["<string>"]={
    "<string>",
    ["<keyword>"]="<value>",
   },
  },
  ["filenames"]={
   ["<string>"]={ "<string>" },
  },
  ["files"]={
   ["list"]={
    ["<string>"]={
     ["name"]="<string>",
     ["style"]="<string>",
     ["weight"]="<string>",
     ["width"]="<string>",
    },
   },
   ["name"]="<string>",
  },
  ["mathematics"]={
   ["alternates"]={
    ["<string>"]={
     ["comment"]="<string>",
     ["feature"]="<hash>",
     ["value"]="<float>",
    },
   },
   ["dimensions"]={
    ["<string>"]={
     ["<unicode>"]={
      ["depth"]="<units>",
      ["height"]="<units>",
      ["width"]="<units>",
      ["xoffset"]="<units>",
      ["yoffset"]="<units>",
     },
    },
   },
   ["italics"]={
    ["<string>"]={
```

```
         ["corrections"]={
          ["<unicode>"]="<float>",
          },
          ["defaultfactor"]="<float>",
          ["disableengine"]="<boolean>",
         },
        },
        ["kerns"]={
         ["<unicode>"]={},
        },
        ["mapfiles"]={ "<string>" },
        ["parameters"]={
         ["<string>"]="<function>",
        },
        ["variables"]={
         ["<string>"]="<value>",
        },
        ["virtuals"]={
         ["<string>"]={
          {
           ["extension"]="<boolean>",
           ["features"]="<hash>",
           ["main"]="<boolean>",
           ["name"]="<string>",
           ["parameters"]="<boolean>",
           ["skewchar"]="<unicode>",
           ["vector"]="<string>",
          },
         },
        },
       },
       ["name"]="<string>",
       ["postprocessors"]={
        ["<string>"]="<function>",
       },
       ["remapping"]={
        ["tounicode"]="<boolean>",
        ["unicodes"]={
         ["<string>"]="<index>",
        },
       },
       ["solutions"]={
        ["experimental"]={
         ["less"]={ "<string>" },
```

```
   ["more"]={ "<string>" },
  },
 },
 ["stylistics"]={
  ["<string>"]="<string>",
 },
 ["typefaces"]={
  ["<string>"]={
   ["boldweight"]="<string>",
   ["features"]="<string>",
   ["fontname"]="<string>",
   ["normalweight"]="<string>",
   ["shape"]="<string>",
   ["shortcut"]="<string>",
   ["size"]="<string>",
   ["width"]="<string>",
  },
 },
 ["version"]="<string>",
}
```

Of course you will never use all the options at the same time. The best place to look for examples are the lfg files in the ConTEXt distribution.[15]

---

[15] At some point we might decide to also support goodies in the generic version.

A Appendix

A.1 The `tfm` file

The (binary) `tfm` file is not human readable but can be turned into a verbose property list which is not that hard to understand.

```
tftopl texnansi-lmr10.tfm
```

Here is an excerpt from the data file. It starts with some general properties of the font. The O means that the value is in octal while the R is a real. Keep in mind that TeX has no datatype 'real' so internally it is just integers representing scaled points.

```
(FAMILY LMROMAN10)
(FACE O 352)
(CODINGSCHEME LY1 ENCODING /TEX'N'ANSI, Y&Y/)
(DESIGNSIZE R 10.0)
(COMMENT DESIGNSIZE IS IN POINTS)
(COMMENT OTHER SIZES ARE MULTIPLES OF DESIGNSIZE)
(CHECKSUM O 4720464277)
```

A text font has the following font dimensions:

```
(FONTDIMEN
    (SLANT      R 0.0)
    (SPACE      R 0.333333)
    (STRETCH    R 0.166667)
    (SHRINK     R 0.111112)
    (XHEIGHT    R 0.43055)
    (QUAD       R 1.0)
    (EXTRASPACE R 0.111112)
    ...
)
```

Kerns and ligatures are packed into a table that is basically a sequence of labelled entries. Here we see the entry for the character f which has three ligatures: ff, fi and fl. Because ligatures can be chained, octal slot 13 will have ligature entries for ffl and ffi.

```
(LIGTABLE
    ...
    (LABEL C f)
    (LIG C f   O 13)
    (LIG C i   O 14)
    (LIG C l   O 10)
```

```
    (KRN O 135 R 0.027779)
    (KRN O  41 R 0.027779)
    (KRN O  51 R 0.027779)
    (KRN O  77 R 0.027779)
    (KRN O 223 R 0.027779)
    (KRN O 224 R 0.027779)
    (KRN O 140 R 0.027779)
    (KRN O  47 R 0.027779)
    (STOP)
    ...
)
```

Each character gets its own entry. In this case there is no depth involved so it is not shown. The comment is just a repetition of the entry in the ligtable.

```
(CHARACTER C f
    (CHARWD R 0.30555)
    (CHARHT R 0.688875)
    (CHARIC R 0.079222)
    (COMMENT
        (LIG C f   O 13)
        (LIG C i   O 14)
        (LIG C l   O 10)
        (KRN O 135 R 0.027779)
        (KRN O  41 R 0.027779)
        (KRN O  51 R 0.027779)
        (KRN O  77 R 0.027779)
        (KRN O 223 R 0.027779)
        (KRN O 224 R 0.027779)
        (KRN O 140 R 0.027779)
        (KRN O  47 R 0.027779)
    )
)
```

A.2 The vf file

A virtual font specification file can be converted to a more readable format with vftovp, for instance:

```
vftovp eurm10.vf
```

The information in a vf file will be combined with the data in the accompanying tfm file so the output looks similar:

```
(VTITLE )
```

```
(FAMILY UNSPECIFIED)
(FACE F MRR)
(CODINGSCHEME TEX MATH ITALIC)
(DESIGNSIZE R 10.0)
(COMMENT DESIGNSIZE IS IN POINTS)
(COMMENT OTHER SIZES ARE MULTIPLES OF DESIGNSIZE)
(CHECKSUM O 24401046203)
(SEVENBITSAFEFLAG TRUE)
```

Because this font is a math font there is no space defined.

```
(FONTDIMEN
    (SLANT     R 0.0)
    (SPACE     R 0.0)
    (STRETCH   R 0.0)
    (SHRINK    R 0.0)
    (XHEIGHT   R 0.459)
    (QUAD      R 1.0)
    (EXTRASPACE R 0.0)
)
```

A virtual font will take glyphs from another font and therefore there are entries that refer to these fonts. In the following definition index 0 is created (the D specifies a decimal entry).

```
(MAPFONT D 0
    (FONTNAME     eurm10)
    (FONTCHECKSUM O 4276740471)
    (FONTAT       R 1.0)
    (FONTDSIZE    R 10.0)
)
(MAPFONT D 1
    (FONTNAME     cmmi10)
    (FONTCHECKSUM O 1350061076)
    (FONTAT       R 1.0)
    (FONTDSIZE    R 10.0)
)
```

The zero indexed font is the default, so in the following entry this font is taken:

```
(CHARACTER C W
    (CHARWD R 0.986)
    (CHARHT R 0.691)
    (CHARIC R 0.056)
    (COMMENT
        (KRN O 177 R  0.056)
```

```
        (KRN O  75 R -0.056)
        (KRN O  73 R -0.083)
        (KRN O  72 R -0.083)
    )
    (MAP
        (SETCHAR C W)
    )
)
```

The next specification is a combination of two other glyphs becoming a new glyph. We see here that the MAP table is actually a sort of program:

```
(CHARACTER O 200
    (CHARWD R 0.622)
    (CHARHT R 0.691)
    (MAP
        (PUSH)
        (MOVEDOWN   R -0.18)
        (MOVERIGHT  R  0.015)
        (SELECTFONT D  2)
        (SETCHAR    O 40)
        (POP)
        (SELECTFONT D  0)
        (SETCHAR    C  h)
    )
)
```

The character information is also in the tfm companion and that is what TeX uses. The virtual information kicks in when the backend is creating the page stream and embedding the fonts.

A.3 The map file

In a map file each line maps a font name onto a file that contains the font shapes in bitmap or outline format. For instance in the file lm-texnansi.map we find the line:

```
texnansi-lmr10 LMRoman10-Regular "enclmtexnansi ReEncodeFont" <lm-texnansi.enc <lmr10.pfb
```

The backend will fetch the glyph data from lmf10.pfb and use the given encoding file to resolve indices to glyph names. A pfb file can contain more than 256 entries so names are used to access the data. The string between quotes is used for the encoding vector in the resulting file.

The second entry in the line is the font name that will be used. This name is also used to control subset behaviour. Multiple references to this name will be collapsed into one

inclusion when possible, thereby making the file as small as possible. You better make sure that the names are unique for a specific font.

In addition to this, there can be directives for extending the font (horizontal stretch) and transforming it into a slanted variant. Both are to be used with care.

In MkIV map files are only used for virtual math fonts and just as in MkII we load such files selectively. Users don't have to worry about this.

A.4 The enc file

For historic reasons, an encoding file is a blob of PostScript, probably because it can be copied into the final output directly. Given that TEX got extended anyway, you can wonder why this information never ended up in an extended `tfm` or `vf` file. It had definitely made the traditional process much more robust.

```
/enclmtexnansi[
/.notdef
/Euro
...
/dotaccent
/hungarumlaut
/ogonek
...
/ffi
/ffl
/dotlessi
/dotlessj
/grave
...
/thorn
/ydieresis
] def
```

There are excactly 256 entries in such a vector and the names should match those in a `pfb` file.

A.5 The afm file

Here we show an excerpt from an `afm` file that comes with Latin Modern Roman. Just as with a `tfm` file we start with some general information. However we don't need to convert the file as is it already in human readable format.

```
StartFontMetrics 2.0
Comment Generated by MetaType1 (a MetaPost-based engine)
```

```
Comment Creation Date: 7th October 2009
Notice Copyright 2003--2009 by B. Jackowski and J.M. Nowacki (on behalf of TeX USERS GROUPS).
Comment Supported by CSTUG, DANTE eV, GUST, GUTenberg, NTG, and TUG.
Comment METATYPE1/Type 1 version by B. Jackowski & J. M. Nowacki
Comment from GUST (http://www.gust.org.pl).
Comment This work is released under the GUST Font License.
Comment For the most recent version of this license see
Comment This work has the LPPL maintenance status `maintained'.
Comment The Current Maintainer of this work is Bogus\l{}aw Jackowski and Janusz M. Nowacki.
Comment This work consists of the files listed in the MANIFEST-Latin-Modern.txt file.
FontName LMRoman10-Regular
FullName LMRoman10-Regular
FamilyName LMRoman10
Weight Normal
ItalicAngle 0
IsFixedPitch false
UnderlinePosition -146
UnderlineThickness 40
Version 2.004
EncodingScheme FontSpecific
FontBBox -430 -290 1417 1127
CapHeight 683.33333
XHeight 430.55556
Descender -194.44444
Ascender 694.44444
Comment PFM parameters: LMRoman10 0 0 0xEE
Comment TFM designsize: 10 (in points)
Comment TFM fontdimen  1: 0          (slant)
Comment TFM fontdimen  2: 3.33333    (space)
Comment TFM fontdimen  3: 1.66667    (space stretch)
Comment TFM fontdimen  4: 1.11111    (space shrink)
Comment TFM fontdimen  5: 4.3055     (xheight)
Comment TFM fontdimen  6: 10         (quad)
Comment TFM fontdimen  7: 1.11111    (extra space)
Comment TFM fontdimen  8: 6.833      (non-standard: uc height)
Comment TFM fontdimen  9: 6.9445     (non-standard: ascender)
Comment TFM fontdimen 10: 11.27      (non-standard: accented cap height)
Comment TFM fontdimen 11: 1.94443    (non-standard: descender depth)
Comment TFM fontdimen 12: 11.27      (non-standard: max height)
Comment TFM fontdimen 13: 2.9        (non-standard: max depth)
Comment TFM fontdimen 14: 5          (non-standard: digit width)
Comment TFM fontdimen 15: 0.88889    (non-standard: uc stem)
Comment TFM fontdimen 16: 12         (non-standard: baselineskip)
Comment TFM fontdimen 17: 0.69444    (non-standard: lc stem)
```

```
Comment TFM fontdimen 18: 0.55556    (non-standard: u, i.e., font unit)
Comment TFM fontdimen 19: 0.22223    (non-standard: overshoot)
Comment TFM fontdimen 20: 0.25       (non-standard: thin stem, hair)
Comment TFM fontdimen 21: 0.30556    (non-standard: cap thin stem, i.e., cap_hair)
Comment TFM headerbyte  9: FontSpecific
Comment TFM headerbyte 49: LMRoman10
Comment TFM headerbyte 72: 234
```

Watch the comments! Because T$_E$X needs a couple of so called fontdimens to be set, the comments list the appropriate values. When a `tfm` file is generated from an `afm` file, these values have to be used.

Each character (or glyph) gets an entry. When we run out of indices i.e. pass the 255 boundary (we start at 0) the index becomes `-1`. Only the width is specified. The height and depth have to be derived from the bounding box for which the specification starts with key `B`.

```
StartCharMetrics 821
...
C  32 ; WX 333.33333 ; N space   ; B  0   0   0   0 ;
...
C 102 ; WX 305.55556 ; N f        ; B 33   0 357 705 ; L f ff ; L i fi ; L k f_k ; L l fl ;
C 105 ; WX 277.77777 ; N i        ; B 33   0 247 657 ;
C 108 ; WX 277.77777 ; N l        ; B 33   0 255 694 ;
...
C  -1 ; WX 500       ; N Acute    ; B 181 493 388 656 ;
C  -1 ; WX 500       ; N acute    ; B 188 510 374 698 ;
C  -1 ; WX 500       ; N acute.ts1 ; B 208 513 392 699 ;
...
EndCharMetrics
```

Watch how this font defines a space character and keep in mind that these fonts date from the time that there was only one kind of space. The `L` entry specifies a ligature.

The names of glyphs are standardized, and even the `f_k` is conforming to standards. This standardization makes it possible to go back from glyphs to characters when copying text from a typeset document.

The kern table is pretty large here and for a reason. First of all the file defines 821 glyphs so the average amount of kerns per glyph is not that large. But take a look at the `A`. Because the `Aacute` has the same shape it kerns in a similar way. This means that ideally all combined characters end up with the same value as their base glyph. However, in our case a bit more selective approach is taken. The `Adieresis` has a different set of kerns, probably to save space. It is for this reason that OpenType fonts have a model of kern classes so that similar shapes can be treated as one when setting kerns. You see a similar issue with ligatures, where often the right part of the shape kerns the same as the (stand alone) first part of the shape does.

```
StartKernData
StartKernPairs 9230
...
KPX seven.prop      hyphen.prop      -37
KPX seven.prop      four.prop        -74
KPX seven.prop      six.prop         -18.5
KPX hyphen.prop     one.prop         -37
KPX hyphen.prop     two.prop         -18.5
KPX hyphen.prop     seven.prop       -55.5
KPX seven.oldstyle  four.oldstyle    -74
KPX A               T                -83.333
KPX Aacute          T                -83.333
KPX Abreve          T                -83.333
KPX Acircumflex     T                -83.333
...
KPX Adieresis       C                -27.778
...
KPX f               bracketright     27.778
KPX f               exclam           27.778
KPX f               parenright       27.778
KPX f               question         27.778
KPX f               quotedblleft     27.778
KPX f               quotedblleft.cm  27.778
KPX f               quotedblright    27.778
KPX f               quotedblright.cm 27.778
KPX f               quoteleft        27.778
KPX f               quoteright       27.778
...
KPX ff              bracketright     27.778
KPX ff              exclam           27.778
KPX ff              parenright       27.778
KPX ff              question         27.778
KPX ff              quotedblleft     27.778
KPX ff              quotedblleft.cm  27.778
KPX ff              quotedblright    27.778
KPX ff              quotedblright.cm 27.778
KPX ff              quoteleft        27.778
KPX ff              quoteright       27.778
...
EndKernPairs
EndKernData
```

If you look closely at the names, you will notice that some glyphs have a variant. In OpenType fonts these variants are grouped in features like oldstyle. The first part of such a name is still part of the standardization, but the second part is up to the font

designer.

The file ends with:

`EndFontMetrics`

A.6 The `otf` file

In the LuaTeX manual you can find an overview of the raw otf format as exposed in a Lua table. The first decade of LuaTeX we used the built-in loader but even then in ConTeXt we didn't use that format directly but used it to create a more compact and efficient table instead. The current release of ConTeXt uses its own loader written in Lua, but the fundamentals have not changed much. The tables are cached and can be read in at high speed. The structure of the tables is unlikely to change much although more data might get added. Although you can access the data it seldom makes sense to do so. Where needed interfaces are provided.

A.7 The `lfg` file

We use the goodies file control what gets added, replaced, patched or manipulated in a font. A goodie file permits us to go beyond what font provide by default. The content of a goodie file differs per font. As we also use this for experiments, not all entries that you find in such files are meant for users.

A.8 Used fonts

The examples in the document depend on the fonts used. Here is a list of fonts used to render this version. Because fonts might have changed in the meantime, some examples might come out other than intended.

filename	`lmtypewriter10-regular.ttf`
instances	2
filesize	128.34 Kb
version	`version 1.011;ps 0.99.3;core 1.0.38;makeotf.lib1.6.5960`

filename	`cambria.ttf`
instances	5
filesize	927.28 Kb
version	`version 5.02a`

filename	`lucidabrightmathot-demi.otf`
instances	3
filesize	166.688 Kb
version	`version 1.801`

filename   lucidabrightmathot.otf
instances   3
filesize   354.176 Kb
version   version 1.801

filename   lucidabrightot.otf
instances   3
filesize   73.284 Kb
version   version 1.801

filename   emojionecolor-svginot.ttf
instances   1
filesize   6686.876 Kb
version   1.0 20160505

filename   ebgaramond12-regular.otf
instances   2
filesize   495.3 Kb
version   version 0.016

filename   hanbatang-lvt.ttf
instances   1
filesize   28032.396 Kb
version   version 1.936; kts build 20131029

filename   husayni.ttf
instances   2
filesize   781.544 Kb
version   version 1.000

filename   ipaexm.ttf
instances   1
filesize   7835.464 Kb
version   version 002.01

filename   punknova-regular.otf
instances   4
filesize   453.944 Kb
version   version 001.003

filename   unifrakturcook.ttf
instances   5
filesize   70.324 Kb
version   version 2012-07-21 ; ttfautohint (v0.9)

filename   uzdr.pfb
instances   2

Appendix

filesize   9.381 Kb
version   001.005

filename   logo10.pfb
instances   2
filesize   0.992 Kb
version   001.002

filename   lmmonoltcond10-regular.otf
instances   1
filesize   64.16 Kb
version   version 2.004;ps 2.004;hotconv 1.0.49;makeotf.lib2.0.14853

filename   lmroman10-bold.otf
instances   2
filesize   111.24 Kb
version   version 2.004;ps 2.004;hotconv 1.0.49;makeotf.lib2.0.14853

filename   lmroman10-italic.otf
instances   3
filesize   118.828 Kb
version   version 2.004;ps 2.004;hotconv 1.0.49;makeotf.lib2.0.14853

filename   lmroman10-regular.otf
instances   16
filesize   111.536 Kb
version   version 2.004;ps 2.004;hotconv 1.0.49;makeotf.lib2.0.14853

filename   texgyredejavu-math.otf
instances   7
filesize   525.008 Kb
version   version 1.106

filename   texgyrepagella-math.otf
instances   3
filesize   601.22 Kb
version   version 1.632

filename   texgyrepagella-regular.otf
instances   15
filesize   144.472 Kb
version   version 2.004;ps 2.004;hotconv 1.0.49;makeotf.lib2.0.14853

filename   xits-math.otf
instances   6
filesize   538.696 Kb
version   version 1.108

filename xits-mathbold.otf
instances 6
filesize 252.74 Kb
version version 1.108

filename xits-regular.otf
instances 1
filesize 253.556 Kb
version version 1.108

filename dejavusans.ttf
instances 1
filesize 741.536 Kb
version version 2.34

filename dejavusansmono-bold.ttf
instances 5
filesize 318.392 Kb
version version 2.34

filename dejavusansmono.ttf
instances 6
filesize 335.068 Kb
version version 2.34

filename dejavuserif-bold.ttf
instances 6
filesize 345.364 Kb
version version 2.34

filename dejavuserif-bolditalic.ttf
instances 1
filesize 336.884 Kb
version version 2.34

filename dejavuserif-italic.ttf
instances 5
filesize 343.388 Kb
version version 2.34

filename dejavuserif.ttf
instances 40
filesize 367.26 Kb
version version 2.34

filename dejavuserifcondensed.ttf
instances 1

filesize 334.04 Kb
version version 2.34

filename seguiemj.ttf
instances 1
filesize 978.292 Kb
version version 1.05

This book is about fonts and how they are dealt with in ConTEXt MkIV and LuaTEX. Although we use ConTEXt as starting point, much applies to the generic font handler that ships as part of this macro package. We discuss the way fonts are dealt with in the engine, font formats, standard features and additional goodies. Tracing and the extensibility of code are also discussed. This book is the more technical companion of the regular ConTEXt font manual.

work in progress