Introduction to OpenType Programming

Goal of this workshop:

- To be able to write elementary layout code in Adobe feature syntax
- To understand the model and principles of OpenType shaping
- To begin to apply these principles to more advanced script requirements

You will need:

- A copy of the OTLFiddle application.

  This also requires you to have the Python "fontTools" library installed, which you can do with either "pip install fontTools" or "easy_install fontTools" from the command line if you don't have it already.
- A copy of `OpenSans-Dummy.ttf` and `OpenSans-Dummy.fea`
- A copy of `Mirza-Dummy.ttf` and `Mirza-Dummy.fea`
- A tool like UnicodeChecker, UnicopediaPlus or a web site like UniView which will help you to paste unfamiliar characters into text.
- [Optionally] A font editor.

Before the workshop you should:

- Download the resources above.
- Install and test OTLFiddle.
- [Optionally] Setup an Arabic keyboard on your computer.
- Have this workbook open in your PDF viewer or editor.

Introductions

Welcome to the workshop! You can find what we're going to be covering in the goals statement above. In the first half of this workshop we are going to be focusing more on understanding the fundamentals of how OpenType features work and how they are processed, rather than looking at specific recipes for doing specific tasks. The aim is that this will give you the tools that you need to solve your own problems in the future and that this knowledge will be transferable to more complex situations. In the second half of the workshop we will demonstrate this by applying these fundamentals to implementing some requirements of an Arabic font.

OpenType programming is a big area, and we cannot cover all of it in one hour! In particular, there are a few important areas which we cannot talk about today:

- creating fonts with language or script specific behavior;
- how we integrate the code that we write into our font production and mastering processes;
- how we can use OpenType layout rules to define glyph categories using the GDEF table;
- how mark positioning and anchors, normally handled by the font editor, are represented in Adobe feature syntax, and how they can be programmed.

If you want more detail on any of these areas, you might find it in my book Fonts and Layout for Global Scripts.

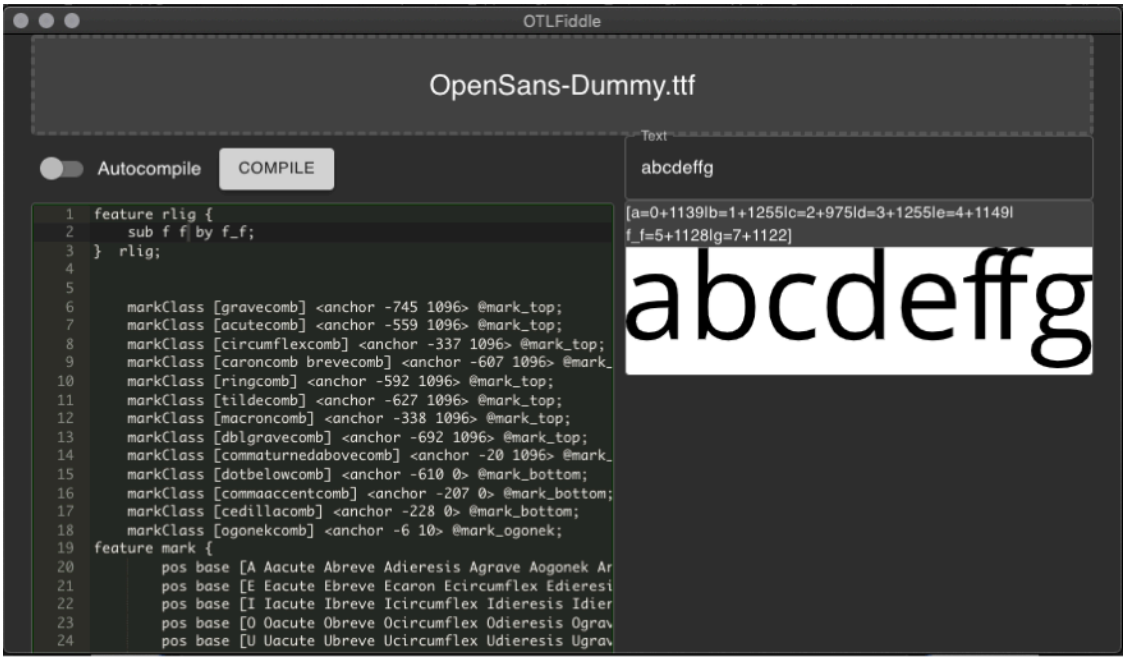
Basic feature coding - substitutions and ligatures

We're going to begin by coding a simple "ff" ligature. Open up OTLFiddle, drop the `OpenSans-Dummy.ttf` font at the top, and paste in the contents of into `OpenSans-Dummy.fea` the editor on the left. (This provides anchor and mark positioning rules for accents).

Then, at the top of the the editor, add the following lines:

```
feature rlig {
  sub f f by f_f;
} rlig;
```

Now hit "Compile", and type some text in the text bar on the right. Your window should now look something like this:



OTLFiddle window

Congatulations! You've made your first OpenType rule. Now I would like you to *experiment* by modifying the rule that you've just made:

- Do we need all those semicolons?
- Are spaces significant?
- Do we really need to write "rlig" *again* at the end?

How about changing the rule itself. We've successfully substituted two glyphs (f f) for one (f_f). Can we substitute:

- One glyph for one?
- More than one glyph for one?
- One glyph for two or more?
- Two glyphs for two?
- A glyph for zero glyphs? (Can you delete a glyph?)

What about using other features rather than `rlig`? Can we substitute glyphs inside a `kern` feature?!

Glyph Classes and Named Classes

Try this rule to turn vowels into their uppercase equivalents. How does it work?

```
feature rlig {
    sub [a e i o u] by [A E I O U];
}  rlig;
```

Answer: When a class is used in a substitution, corresponding members are substituted on both sides.

Experiment: What happens when a glyph class is only used on the left hand side? Try it and find out:

```
feature rlig {
    sub f [a e i o u] by f_f;
}  rlig;
```

Named Glyph Classes

We can *define* a glyph class, naming a set of glyphs so we can use the class again later:

```
@lower_vowels = [a e i o u];
@upper_vowels = [A E I O U];
```

Now anywhere a glyph class appeared, we can use a named glyph class instead (including in the definition of other glyph classes!):

```
@vowels = [@lower_vowels @upper_vowels];

feature rlig {
  sub @lower_vowels by @upper_vowels;
} rlig;
```

How OpenType shaping works

Let's look at our first example again:

```
feature rlig {
  sub f f by f_f;
} rlig;
```

We can can write this explicitly in terms of lookups, like this:

```
feature rlig {
  lookup rlig_1 {
    sub f f by f_f;
  } rlig_1;
} rlig;
```

You can also define a lookup separately, and refer to it within a feature, like this:

```
lookup rlig_1 {
    sub f f by f_f;
} rlig_1;

feature rlig {
    lookup rlig_1;
} rlig;
```

What difference does it make?

Experiment: Consider the difference between this:

```
feature rlig {
    sub a by b;
    sub b by c;
} rlig;
```

and

```
feature rlig {
    lookup l1 { sub a by b; } l1;
    lookup l2 { sub b by c; } l2;
} rlig;
```

Try each one out in OTLFiddle against the input string “abc”. What happened? Can you work out why? (Hint: How many lookups in the feature in each case? How many rules in each lookup?)

Ordering of rules, lookups and features

Experiment: Try changing the order of the rules:

```
feature rlig {
    sub f f by f_f;
    sub f f i by f_f_i;
    sub f l by f_l;
} rlig;
```

Does it make a difference?

Experiment: What does the following code do on the string `abc`?

```
feature rlig { sub a by b; } rlig;

feature ccmp { sub b by c; } ccmp;

feature rlig { sub c by d; } rlig;
```

Guess first before trying it out! Will it produce:

- `bcd`
- `bdd`
- `ccd`
- `ddd`
- Something else?

What about this?

```
feature ccmp { sub b by c; } ccmp;

feature rlig { sub a by b; } rlig;

feature rlig { sub c by d; } rlig;
```

Remember: shapers use *features* to gather *lookups*.

Lookup Flags

Experiment: Let's go back to our original ligature and try it against some new text:

```
feature rlig {
  lookup ff_ligature {
    sub f f by f_f;
  } ff_ligature;
} rlig;
```

The text I want you to try this against is f̊f. (Maybe you want to copy and paste that into OTLFiddle.) It is the string "f", COMBINING RING ABOVE (Unicode codepoint U+030A), "f".

Try this now:

```
feature rlig {
  lookup ff_ligature {
    lookupflag IgnoreMarks;
    sub f f by f_f;
  } ff_ligature;
} rlig;
```

Now we get the ligature *and* the accent.

Positioning Phase

Experiment: Remove the features you have added in OTLFiddle, and paste in this one:

```
feature kern {
    lookup adjust_f {
        pos f <0 0 200 0>;
    } adjust_f;
} kern;
```

- Try it on text like `afbffc`. What did it do?
- Play about with each of the components of the value record. How do they affect the output?
- What about *negative* values?
- Try to turn the "f" into an "accent" - with no width of its own, positioned on top of the letter before it. What happens if you type two f's in a row? (Can you work out why?)

Types of rule

We've seen a variety of substitution rules, as well as a positioning rule. Are there any other kinds of positioning rule? Are there any other rules? As it happens, the whole set of rule types we need to know about are:

- Substitution
  - One to one
  - Many to one (ligature)
  - One to many
  - Reverse chaining (Only used for Nastaliq fonts)
- Positioning
  - Single positioning: `pos glyph <…>;`
  - Pair positioning: `pos glyph1 <…> glyph2;`
- Anchor
- Chain

Some *questions*:

- Why isn't there a many to many substitution rule? (Hint: `IgnoreMarks`.)
- Why do you think a pair positioning rule might be useful?

Anchor rules (of which there are four kinds: cursive attachment, mark positioning, mark-to-mark and mark-to-ligature attachment) are for sticking a glyph onto another glyph. Again, unfortunately we don't have time to get into them today.

Chain rules, however, are a lot of fun. They allow us to call a lookup at a given position in the glyph stream, if a certain set of conditions are met. Let's try one first, and then we'll explain it later.

Experiment: Paste this into OTLFiddle:

```
lookup ff_ligature {
    sub f f by f_f;
} ff_ligature;

feature rlig {
    sub [A E I O U] f' lookup ff_ligature;
} rlig;
```

Now try it on the string `Off off`. What happens? Why?

Introduction to Arabic font engineering

Now let's load the `Mirza-Dummy.ttf` font and paste in the rules from `Mirza-Dummy.fea` into OTLFiddle. The rules I've provided do two things: basic Arabic shaping using the `init/medi/fina` features, and mark positioning. These are things normally done for you by the font editor, so I don't expect you to code them yourself.

Now I have some challenges for you!

- There is a required ligature in Arabic between the glyphs `lam-ar` and `alef-ar`. These should be replaced by `lam_alef.ar` in isolated form, and `am_alef-ar.fina` when in final form. (Alef is "right-joining", meaning it won't connect to anything on its left, so there's no medial form.) Of course, the ligature should work even if there are marks like `kasra-ar` (U+0650) or `fatha-ar` (U+064E) attached to either the lam or the alef.

You will know you have completed this challenge when the input text لِا سلاً is shaped as `[fatha-ar=9@53,-144|lam_alef-ar.fina=9+559|seen-ar.init=7+530|space=6+200|kasra-ar=0@335,136|lam_alef-ar=0+447]`.

Note that even though Arabic is read and rendered right-to-left, the OpenType glyph stream is in *logical order* i.e. in لا the `lam-ar` comes first.

- Similarly the `kaf-ar lam-ar` ligature `kaf_lam-ar` needs to work in all four forms (`kaf_lam-ar`, `kaf_lam-ar.init`, `kaf_lam-ar.medi`, `kaf_lam-ar.fina`) - and of course with potential marks.
- Shape the text پے (`peh-ar.init yehbarree-ar.fina`). Notice that the dots of the peh clash with the bar of the yeh barree. Write a rule which inserts a `kashida-ar` glyph after `peh-ar.init` and `peh-ar.medi` when the next glyph is `yehbarree-ar.fina`.

Hint: You can't directly substitute `peh-ar.init yehbarree-ar.fina` with `peh-ar.init kashida-ar yehbarree-ar.fina` - that would be a many-to-many substitution which OpenType doesn't support. You're going to need to split this into two rules: one which checks that the context is right and chains to another rule, which makes a one-to-many substitution.

- Shape the text سپے بپے پے. Notice that the sequence `[beh-ar.medi beh-ar.init yeh-farsi.init yeh-farsi.medi] kasra-ar yehbarree-ar.fina` makes the `kasra-ar` clash into the `yehbarree-ar.fina`. Add a chained positioning rule in the `kern` feature to reposition kasra in this context below the yeh barree.

- Bonus challenge (to take home): Aya ("verse") numbers in the Quran are *enclosed* in a decorative border. When one- to three-digit sequences of Arabic numbers (`one-ar two-ar ...`) are followed by U+06DD END OF AYA SIGN, they should be replaced by small numbers (`one-ar.small two-ar.small ...`). You will need a chained substitution rule to achieve this. They should *also* have a chained *positioning* rule which reduces the advance width of each number to zero, and then displaces them so that they appear centered inside the `endofayah-ar`. Do this with a one-digit number first before working up to two and three digits!

Close-out

Thank you for attending the workshop. This is the first time it has been run, so please let me know of any feedback you have. I hope it has been useful for you.