OpenType Feature File Specification

Version

[Document version 1.11. Last updated 4 Sept 2012]

Caution:

Portions of the syntax unimplemented by Adobe are subject to change.

1. Introduction

An OpenType feature file is a text file that contains the typographic layout feature specifications for an OpenType font in an easy-to-read format. It may also contain override values for certain fields in the font tables. It is read in during the creation or editing of an OpenType font. This document specifies the feature file grammar.

This is an example of a complete feature file (keywords are shown boldface):

```
# Script and language coverage
languagesystem DFLT dflt;
languagesystem latn dflt;

# Ligature formation
feature liga {
    substitute f i by f_i;
    substitute f l by f_l;
} liga;

# Kerning
feature kern {
    position A Y -100;
    position a y -80;
    position s f' <0 0 10 0> t;
} kern;
```

This file specifies the formation of the "f_i" and "f_l" ligatures, and the kern values of the glyph pairs "A" "Y" and "a" "y". It also specifies a contextual positioning adjustment for "f" when preceded by "s" and followed by "t". It also specifies that all features will be applied under all languages in the latn script, and for all scripts not named in the feature file.

Note: all "Implementation Notes" and "Currently not implemented" comments in the rest of the specification below refer to the Adobe implementation of the feature file grammar in the makeotf program, unless otherwise indicated.

2. Syntax

2.a. Comments

The "#" character indicates the start of a comment; the comment extends until the end of the line. Text on a line after the comment is discarded before processing.

2.b. White space

White space is not significant except for delimiting tokens. You can have multiple line endings, spaces, and tabs between tokens. Macintosh, UNIX and PC line endings are all supported.

2.c. Keywords

This is a complete list of keywords in the feature file language, shown in **boldface** in examples:

anchor
anchorDef
anonymous (or anon)
by
contour
cursive
*device
enumerate (or enum)
excludeDFLT (deprecated)
exclude_dflt
feature block; feature statement
from
ignore substitute; ignore position
IgnoreBaseGlyphs
IgnoreLigatures
IgnoreMarks
MarkAttachmentType
UseMarkFilteringSet
include
includeDFLT (deprecated)
include_dflt
language
languagesystem
lookup block and statement
lookupflag
mark
markClass
nameid
NULL device; NULL value record; NULL anchor
parameters
position (or pos)
*required
RightToLeft
reversesub (or rsub)
script
substitute (or sub)

```
subtable
table
useExtension
valueRecordDef
```

The following are keywords only in their corresponding table/feature blocks, and are shown in **boldface** in examples:

```
HorizAxis.BaseTagList    # BASE table
HorizAxis.BaseScriptList  "
*HorizAxis.MinMax         "
VertAxis.BaseTagList      "
VertAxis.BaseScriptList   "
*VertAxis.MinMax          "
GlyphClassDef           # GDEF table
Attach                  "
*LigatureCaretByDev       "
LigatureCaretByIndex      "
LigatureCaretByPos        "
MarkAttachClass           "
FontRevision             # head table
CaretOffset              # hhea table
Ascender                 # hhea table
Descender                # hhea table
LineGap                  # hhea table
Panose                  # OS/2 table
TypoAscender              "
TypoDescender             "
TypoLineGap               "
winAscent                 "
winDescent                "
UnicodeRange              "
CodePageRange             "
XHeight                 "
CapHeight                 "
Vendor                  "
sizemenuname             # size feature
VertTypoAscender         # vhea table
VertTypoDescender         "
VertTypoLineGap           "
VertOriginY             # vmtx table
VertAdvanceY             # vmtx table
```

[Currently not implemented.]*

The following are keywords only where a tag is expected:

```
DFLT  # can be used only with the script keyword and as the script value with the languagesystem keyword.
dflt  # can be used only with the language keyword and as the language value with the languagesystem keyword.
```

The only permitted language tag for the 'DFLT script is 'dflt'.

2.d. Special characters

```
#    pound sign     Denotes start of comment
;    semicolon      Terminates a statement
,    comma          Separator in various lists
@    at sign        Identifies glyph class names
\    backslash      Identifies CIDs. Distinguishes glyph names from an identical keyword
-    hyphen         Denotes glyph ranges in a glyph class
=    equalsign      Glyph class assignment operator
'    single quote   Marks a glyph or glyph class for contextual substitution or positioning
" "  double quotes   Enclose a name table string
{ }  braces         Enclose a feature, lookup, table, or anonymous block
[ ]  square brackets Enclose components of a glyph class
< >  angle brackets  Enclose a device, value record, contour point, anchor, or caret
( )  parentheses     Enclose the file name to be included
```

2.e. Numbers and other metrics

2.e.i. Number

A <number> is a signed decimal integer (without leading zeroes). For example:

```
-150
1000
```

It is used in device tables [§2.e.iii] and contour points [§2.e.vi], as well as the values of various table fields [§9].

2.e.ii. Metric

A <metric> value is simply a <number> in font design units. It is used in value records [§2.e.iv] for positioning rules, as well as to express the values of various table fields [§9].

[Note: Multiple master support has been withdrawn as of OpenType specification 1.3.]

2.e.iii. Device table

[Currently not implemented.]

A <device> represents a single device table or a null offset to it. It is used in value records [§2.e.iv], anchors [§2.e.vii], and and the GDEF table LigatureCaret statements [§9.b].

- Device format A:

This specifies a comma-separated list of <number> pairs. The first <number> in a pair represents a ppem size and the second the number of pixels to adjust at that ppem size:

< **device** <number> <number> # ppem size, number of pixels to adjust
(, <number> <number>)* > ; # zero or more such pairs

For example:

<**device** 11 -1, 12 -1> # Adjust by -1 at 11 ppem and 12 ppem

- Device format B, the null device:
- `< device NULL >`

This format is used when an undefined <device> is needed in a list of <device>s.

2.e.iv. Value record

A <valuerecord> is used in some positioning rules [§6].

It must be enclosed by angle brackets, except for format A, in which the angle brackets are optional. Note that the <metric> adjustments indicate values (in design units) to add to (positive values) or subtract from (negative values) the placement and advance values provided in the font (in the 'hmtx' and 'vmtx' tables).

- Value record format A:
- `<metric> # Angle brackets around value are not allowed.`

Here the <metric> represents an×advance adjustment, except when used in the 'vkrn' feature, in which case it represents a Y advance adjustment. All other adjustments are implicitly set to 0. This is the simplest feature file <valuerecord> format, and is provided since it represents the most commonly used adjustment (i.e. for kerning). For example:

-3 # without <>

- Value record format B:
- `< <metric> <metric> <metric> <metric> >`

Here, the <metric>s represent adjustments for×placement, Y placement,×advance, and Y advance, in that order. For example:

<-80 0 -160 0> #×placement adjustment: -80;× advance adjustment: -160

- Value record format C: *[Currently not implemented.]*
- `< <metric> <metric> <metric> <metric> <device> <device> <device> <device> >`

Here, the <metric>s represent the same adjustments as in format B. The <device>s represent device tables [§2.e.iii] for× placement, Y placement,×advance, and Y advance, in that order. This format lets the editor express the full functionality of an OpenType value record. For example:

<-80 0 -160 0 <**device** 11 -1, 12 -1>
<**device NULL**>

<**device** 11 -2, 12 -2>
<**device NULL**>>

This example specifies adjustments for×placement and×advance, as well as device adjustments at 11 and 12 ppem sizes for×placement and×advance.

- Value record format D, the null value record: *[Currently not implemented.]*

<**NULL**>

Value record not defined

- Value record format E, the named value record:

  <name>.

  For example:

  < KERN_POS_1 >

  The name must have been defined with a **valueRecordDef** statement before being used.

2.e.v. Named value record definition.

The **valueRecordDef** keyword is used to define a named value record. This name can then be used in value records instead of coordinates. It offers the advantage of being able to change the coordinates in the named value record definition only, and having that single edit change the coordinates used in all the rules in which the named value record is used. The format is:

valueRecordDef <coordinates> <name>;

where the coordinates can be in value record formats A or B. The anchor name follows the same rules as are used to form glyph names. For example:

valueRecordDef -10 FIRST_KERN;
valueRecordDef 0 0 20 0 SECOND_KERN;

These named value coordinates can then be used in value records. For example:

pos T V < SECOND_KERN > ;

Note than when the value record name is used, it must be enclosed by angle brackets, whether it is a single value or four value record.

2.e.vi. Contour point

A <contour point> is used in anchors [§2.e.vii] and the GDEF table LigatureCaret statements [§9.b]. It takes the format:

contourpoint <number>

where <number> specifies a contour point index. For example:

contourpoint 2

Note: Since CFF OpenType fonts do not specify contour point indexes, a <contour point> may be used only with TrueType OpenType fonts.

2.e.vii. Anchor

An <anchor> is used in some positioning rules [§6]. It takes 5 formats:

- Anchor format A:

  < **anchor** <metric> <metric> > #xcoordinate, Y coordinate

  For example:

  <**anchor** 120 -20>

- Anchor format B:

  <**anchor** <metric> <metric> #xcoordinate, Y coordinate
  <contour point> >

  For example:

  <**anchor** 120 -20 **contourpoint** 5>

- Anchor format C:

  <**anchor** <metric> <metric> # X coordinate, Y coordinate
  <device><device> > # X coord device, Y coord device

  For example:

  <**anchor** 120 -20 <**device** 11 1> <**device NULL**>>

- Anchor format D, the null anchor:

  <**anchor NULL**> # Anchor not defined

- Anchor format E, the named anchor:

  <**anchor** <name> >

  For example:

  <**anchor** TOP_ANCHOR_1>

An anchor name must be defined before it is used - see the following section on the **anchorDef** keyword.

2.e.viii. Named anchor definition

The **anchorDef** keyword is used to define a named anchor. This name can then be used in anchor definitions instead of coordinates. It offers the advantage of being able to change the coordinates in the named anchor definition only, and having that single edit change the coordinates used in all the rules in which the named anchor is used. The format is:

anchorDef <coordinates> <name>;

where the coordinates can be in Anchor Format A or B. The anchor name follows the same rules as are used to form glyph names. For example:

anchorDef 300 0 ANCHOR_1;
anchorDef 120 -20 **contourpoint** 5 ANCHOR_2;

These named anchors can then be used in anchor definitions. For example:

<**anchor** ANCHOR_2 >

2.f. Glyphs

These are represented by one of:

2.f.i. Glyph name

A glyph name may be up to 31 characters in length, must be entirely comprised of characters from the following set:

A-Z a-z 0-9 . (period) _ (underscore)

and must not start with a digit or period. The only exception is the special character ".notdef".

"twocents", "a1", and "_" are valid glyph names. "2cents" and ".twocents" are not.

An initial backslash serves to differentiate a glyph name from an identical keyword in the feature file language. (See §2.c for a list of keywords.) For example, a glyph named "table" must be specified in the feature file as:

\table

A glyph name alias database may be used by the implementation of the feature file grammar. If it is used, then it is the responsibility of the implementation to correlate the glyph name aliases used in the feature file with the actual glyph names in the font.

2.f.ii. CID

CIDs are represented by a non-negative <number> [§2.e.i] preceded by a backslash. For example:

\101
\0

2.g. Glyph classes

Note: The feature file glyph classes described in this section are not to be confused with glyph classes of OpenType Layout ClassDefs. The latter are described in the chapter "Common Table Formats" in the OpenType Font File Specification.

A feature file glyph class, <glyphclass>, represents a single glyph position in a sequence and is denoted by a list of glyphs enclosed in square brackets. For example:

[endash emdash figuredash]

An example of a sequence which contains a glyph class is:

space [endash emdash figuredash] space

This would match any of the 3 sequences "space endash space", "space emdash space", or "space figuredash space" during OpenType layout.

A feature file glyph class that contains only one single glyph is known as a singleton glyph class.

A feature file glyph class is also used to represent the set of alternate glyphs in an alternate substitution lookup type rule.

2.g.i. Ranges

A range of glyphs is denoted by a hyphen:

[<firstGlyph> - <lastGlyph>]

Spaces around the hyphen are not required since hyphens are not permitted in feature file glyph names. For example:

```
[\0-\31]
[A-Z]
```

For CID fonts, the ordering is the CID ordering.

For non-CID fonts, the ordering is independent of the ordering of glyphs in the font. <firstGlyph> and <lastGlyph> must be the same length and can differ:

1. By a single letter from A-Z, either uppercase or lowercase. For example:

   ```
   [A.swash - Z.swash]
   [a - z]
   ```

   The range is expanded by incrementing the letter that differs, while keeping the rest of the glyph name the same.

2. By up to 3 decimal digits in a contiguous run. For example:

   ```
   [ampersand.01 - ampersand.58]
   ```

   The range is expanded by incrementing the number values, while keeping the rest of the glyph name the same.

   ```
   [ampersand.1 - ampersand.58]        # invalid
   ```

   is not a valid glyph class since the length of the glyph names differ.

Note that

```
[zero - nine]
```

is not a valid glyph range, as the intended range is not in alphabetic order. It must be enumerated explicitly:

```
@digits = [zero one two three four five six seven eight nine];
```

2.g.ii. Named glyph classes

A glyph class can be named by assigning it to a glyph class name, which begins with the "@" character, and then referred to later on by the glyph class name. For example:

```
@dash = [endash emdash figuredash];     # Assignment
space @dash space                       # Usage
```

The part of the glyph class name after the "@" is subject to the same name restrictions that apply to a glyph name, except that its maximum length is 30.

Glyph class assignments can appear anywhere in the feature file. A glyph class name may be used in the feature file only after its definition.

When a glyph class name occurs within square brackets, its elements are simply added onto the other elements in the glyph class being defined. For example:

```
@Vowels.lc = [a e i o u];
@Vowels.uc = [A E I O U];
@Vowels = [@Vowels.lc @Vowels.uc y Y];
```

Here the last statement is equivalent to:

```
@Vowels = [a e i o u A E I O U y Y];
```

No square brackets are needed if a glyph class name is assigned to another single glyph class name. For example:

```
@Figures_lining_tabular = @FIGSDEFAULT;
```

Ranges, glyphs, and glyph class names can be combined in a glyph class. For example:

```
[A.oldstyle - Z.oldstyle ampersand.oldstyle  @smallCaps]
```

Implementation Note: When feature file glyph sequences (including glyph classes) are converted into OpenType Layout ClassDefs or Coverages in the font, the Adobe implementation ensures that ClassDefs or Coverages that are identical are shared, even if they are in different features. This happens regardless of whether ranges, glyphs or glyph class names were used to express the feature file glyph classes. (The only exception to this is for lookups that use the Extension lookup types: such lookups will not share their ClassDefs and Coverages with non-extension lookups.)

2.h. Tags

Tags are four-letter identifiers. These are denoted simply by tag name, without any final spaces, and are distinguished from glyph names by context. For example:

```
DEU
```

Note that the final space in the example is implicit.

The special language tag 'dflt' denotes the default language system of the corresponding script.

2.i. Lookup block labels

The same length and name restrictions that apply to a glyph name apply to a lookup block label.

3. Including files

Including files is indicated by the directive:

```
include(<filename>);
```

The implementation software is responsible for handling the search paths for the location of the included files.

In a typical implementation, if the file name were absolute then that path would be used. If the file name were relative, then it would be appended to the directory of the including feature file.

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

A maximum include depth of 5 ensures against infinite include loops (files that include each other).

4. Specifying features

4.a. feature

Each feature is specified in a feature block:

```
feature <feature tag> {
  # specifications go here
} <feature tag>;
```

For example:

```
feature liga {
  # ...
} liga;
```

The 'aalt' feature is treated specially; see §8.a. For example, the "useExtension" keyword may optionally precede "{" in its feature block syntax, and other features can be referred to with a "feature" statement within its feature block. The 'size' feature is also treated specially; see §8.b.

A feature file "rule" is a statement that specifies glyph substitution or glyph positioning. A feature block may contain glyph substitution rules [§5], glyph positioning rules [§6], or both.

A lookup is a group of rules of the same type. See §4.e.

4.b. Language system

An OpenType language system is any combination of a script tag and a language tag. (In the text of this document, the notation <script tag>/<language tag> is used to refer to a language system; for example, 'script latn;'/'language dflt' denotes the default language of the Latin script.)

The lookups in every OpenType feature must be registered under one or more language systems. The lookups of a particular feature may vary across the language systems under which the feature is registered.

There are two ways to specify language system in the feature file: with the languagesystem keyword outside of feature definition blocks, and by the script and language keywords within feature definition blocks.

4.b.i. languagesystem

In practice, most or all of the features in a font will be registered under the same set of language systems, and a particular feature's lookups will be identical across the language systems under which the feature is registered.

The "languagesystem" statement provides a simple directive to use in this case. It is the simplest way to specify language system in the feature file. (For the 'aalt' and 'size' features, it is the only way to specify language system.) One or more such statements may be present in the feature file at global scope (i.e. outside of the feature blocks or any other blocks) and before any of the feature blocks:

```
languagesystem <script tag> <language tag>;
```

When these statements are present, then all the lookups in each feature that does not contain an explicit "script" or "language" statement (see 4.b.ii below) will be registered under every language system specified by the "languagesystem" statement(s). If a feature block does contain 'script' or 'language' tags, then all lookups that occur before the first 'script' or 'language' tag will also be applied under all the specified languagesystems.

If no "languagesystem" statement is present, then the implementation must behave exactly as though the following statement were present at the beginning of the feature file:

```
languagesystem DFLT dflt;
```

If any languagesystem statement is used, then the statement specifying:

languagesystem DFLT dflt;

must be specified explicitly; if not, this languagesystem will not be included in the font. This script/language pair is special: it is used if a program cannot find a match in the font to the current writing script and language. If it is not in your font, then all the rules may be invisible to the program if your font does not have a match for the current script and language. It is strongly recommended to use the statement 'languagesystem DFLT dflt;'.

If the statement 'languagesystem DFLT dflt;' is present, it must be the first of the languagesystem statements. Note that the only permitted language tag for the 'DFLT' script is 'dflt'.

Please see example 1 in §4.g below.

4.b.ii. script and language

Occasionally a feature may need to be specified whose lookups vary across the language systems of the feature, or whose language systems vary from the set of language systems of the rest of the features in the file, as specified by the "languagesystem" statements). In these cases, "script" and "language" statements will need to be used within the feature block itself. Such statements affect only that feature.

Rules that are specified after the start of a feature and before the first "script" and/or "language" statement will be included in all the language systems specified by the "languagesystem" statements. If you do not want any of the rules in the feature to be registered under the language systems specified by the "languagesystem" statements, then a "script" and/or "language" statement for a script must be present before the first rule in the feature.

Once the first script or language statement occurs within a feature block, subsequent lookups and rules are registered only within the currently specified script and language. To register a rule or lookup under more than one script and language, you must explicitly include it following each script and language specification.

The one exception to this rule are the default lookups. There are two levels of default lookups. Rules specified between the start of a feature definition and the first 'script' are added to all language-systems, unless a language statement specifies the 'exclude_dflt' keyword. Rules specified between the occurrence of the 'script' statement and the first language statement other than 'dflt' are added to explicitly specified languages for the current script, but not to other scripts, nor to other languages of the same script that are not named in the feature. If your font has several languages for a given script, and you need language specific rules for only some of the languages, you should still explicitly name all of the languages so that they will inherit the script-level default rules.

The current script and language attributes may be changed as follows:

- **"script" statement:**

  script
  <script tag>;

  For example:

  script kana;

  When a "script" statement is seen, the language attribute is implicitly set to 'dflt', and the lookupflag attribute is implicitly set to 0. The script attribute stays the same until explicitly changed by another "script" statement or until the end of the feature.

- **"language" statement:**

  The language attribute stays the same until explicitly changed, until the script is changed, or until the end of the feature. To change the language attribute, use the "language" statement:

  language
  <language tag> [**exclude_dflt**|**include_dflt**] [**required**];

  To exclude a set of rules from only one or a few languages, you must define the set of rules as a lookup , and explicitly include the lookup in under the languages that should include it, and omit it from the rules included under the languages where it should be excluded.

  Note: the excludeDFLT and includeDFLT keywords still work, but are deprecated and will cause a warning to appear.

  Here is an example statement:

  language DEU;

  As a result of this statement, (a) the language attribute is changed to 'DEU ', and (b) the current default lookups are automatically included into the language system specified by the current script and language DEU.

  If (b) is not desired, as may occasionally be the case, then the keyword "exclude_dflt" must follow the language tag. For example:

  language DEU **exclude_dflt**;

  The keyword "include_dflt" may be used to explicitly indicate the default default lookup-inheriting behavior. For example:

  language DEU **include_dflt**;
  # Same as: language DEU;

  The keyword "required", when present, specifies the current feature as the required feature for the specified language system.

  [The keyword "required" is currently not implemented.]

Since the 'aalt' and 'size' features are treated specially, "script" and "language" statements are not allowed within these features.

Special notes:

- 'DFLT' is a valid value for the 'script' tag, and 'dflt' is not.
- 'dflt' is a valid value for the 'language' tag, and 'DFLT' is not.
- The DFLT script cannot have any language systems other then 'dflt'.
- There is no such thing as a 'dflt' language tag in the actual OpenType data structures; the data structures hold these rules in a special record, rather than referencing them from the list of language tags for the current script. The 'dflt' language tag is just a convenience in the feature file syntax for setting the current language to be the default language system.

Please see §4.g below for detailed examples.

4.c. parameters

The "parameters" statement specifies the feature parameters for the currently defined language system. It is currently supported only for the 'size' feature; see §8.b.

4.d. lookupflag

The chapter "Common Table Formats" in the OpenType Font File Specification describes the LookupFlag field in the Lookup table.

The lookupflag attribute defaults to 0 at the start of a feature block.

The lookupflag attribute stays the same until explicitly changed, until a lookup reference statement is encountered that changes it, until the script is changed, or until the end of the feature.

To change the lookupflag attribute explicitly, use the lookupflag statement, which takes two formats:

- lookupflag format A:

  lookupflag <named lookupflag value> (, <named lookupflag value>)*;

  Here, the individual lookup flag values to be set are expressed in a comma-separated list of one or more <named lookupflag value>s, in no particular order. A <named lookupflag value> is one of the following:

  RightToLeft
  IgnoreBaseGlyphs
  IgnoreLigatures
  IgnoreMarks
  MarkAttachmentType <glyph class name>
  UseMarkFilteringSet <glyph class name>

  At most one of each of the above 6 kinds of <named lookupflag> values may be present in a lookupflag statement. For example, to skip over base glyphs and ligature glyphs:

  lookupflag IgnoreBaseGlyphs IgnoreLigatures;

  Base, ligature, and mark glyphs are specified in the glyph class definition of the GDEF table.

  To skip over all mark glyphs except for those of mark class @TOP_MARKS:

  lookupflag MarkAttachmentType @TOP_MARKS;

  The class name used with MarkAttachmentType can be either a regular glyph class name or a mark class name. The glyph sets of the referenced classes must not overlap, and the MarkAttachmentType statement can reference at most 15 different classes.

  A new lookup flag which is proposed for OpenType spec 1.6, **`UseMarkFilteringSet`**, is also supported. This works the same as the MarkAttachmentType, but allows you to use up to 16K different mark classes, and allows the glyph sets of the referenced classes to overlap. The drawback to this lookupflag is that as of Nov 2008, there are not yet any programs that support it.

- lookupflag format B:

  lookupflag <number>;

  Here the entire lookup flag value is specified simply as a <number>. The format A example above could equivalently be expressed as:

  lookupflag 6;

  Format A is clearly a superior choice for human readability when the lookupflag value is not 0. However, a lookupflag value of 0 can be set only with format B, not with format A:

  lookupflag 0;

The base glyphs, ligatures, and mark classes are defined in the GlyphClassDef of the GDEF table block [§9.b].

4.e. lookup

A lookup is a group of rules of the same type. The font editor can label a run of rules and refer to it explicitly later on, in

order to have different parts of the font tables refer to the same lookup. This decreases the size of the font in addition to freeing the editor from maintaining duplicate sets of rules.

A lookup in the OpenType font will be created from each named lookup block or each run of rules with the same feature, script, language, lookupflag and lookup type attribute.

To define and label a lookup, use a named lookup block:

```
lookup <label> [useExtension] {
  # rules to be grouped
} <label>;
```

A named lookup block may be defined either inside or outside of a feature block. In either case, it may be referenced in different feature blocks. If it is defined outside a feature block, is is referred to as a 'stand-alone' lookup.

The lookup will be created with a GSUB or GPOS Extension lookup type if and only if the optional "useExtension" keyword is used.

A lookup block may be defined either inside or outside of feature blocks. You may not use the **script** or **language** keywords within a named lookup block.

The useExtension keyword has two effects: all the records of all types that are referenced by a lookup qualifier placed in one contiguous block of data, and the offset to the lookup may be 32 bits rather limited to 16 bits.

When you font cannot be built because of an offset overflow error (meaning that the offset from one record to another record exceeds the 64 Kbyte limit imposed by the maximum size possible for a 16-bit offset field), then add this qualifier to the largest lookup. Keep adding it to more lookups until your font will build.

Note that since the Extension lookup types were added in OpenType specification v1.3, they will not be recognized by all OpenType layout parsers.

(See also §8.a for how to specify the entire 'aalt' feature be made with the Extension lookup type.)

To refer to the lookup later on, use a lookup reference statement:

```
lookup <label>;
```

For example:

```
lookup SHARED{      # lookup definition
  # ...
} SHARED;
# ...
lookup SHARED;      # lookup reference
```

An example of a lookup that uses the Extension lookup type:

```
lookup EXTENDED_KERNING useExtension {      # lookup definition
  # ...
} EXTENDED_KERNING;
# ...
lookup EXTENDED_KERNING;    # lookup reference. "useExtension" not needed
```

Since the labeled block literally defines a single lookup in the font, the rules within the lookup block must be of the same lookup type and have the same lookupflag attribute. A lookup block may not contain any other kind of block. The order of lookups within a font is defined by the order of the lookup definitions in the feature file.

For contextual rules, the rules of the lookup are ordered in the font file in the same order that they are written in the feature file. For non-contextual rules, the implementation sorts the rules to avoid conflict; for example, the ligature substitution rule for f_f_i will be written before the ligature substitution rule for f_i, no matter what their order is in the feature file. .

4.f. markClass

The markClass keyword is used to identify a mark glyph class definition statement.

A mark glyph class name is defined differently than a regular glyph class. The mark class definition is built up by a one or more of statements in the form:

```
markClass <glyph|glyphclass> <anchor>  <mark glyph class name>;
```

Each additional mark statement for a mark class adds the referenced glyphs to that mark class.

The <anchor> [§2.e.vii] indicates the point on the mark glyph(s) by which it is attached to a matching anchor point on a base glyph. If a mark glyph has an anchor point at <anchor 300, 0> and the base glyph has an anchor point at <anchor 400 300>, then the mark glyph will be shifted so that the point x=300, y = 0 in its design space will be superimposed on the pointx= 300, y = 400 in the design space of the base glyph.

For example:

```
markClass [acute grave dieresis] <anchor 350 0>  @MARK_TOP_ACCENTS;
```

If all the mark glyphs which belong to a mark class have the same anchor, then the mark class can be defined with a single statement, as above. However, a single mark statement can define only a single anchor point, so when glyphs in a mark class have different anchor points, more than one mark statement must be used to define the mark class. For example:

```
markClass [acute grave] <anchor 350 0>  @MARK_TOP_ACCENTS;
markClass [dieresis umlaut] <anchor 400 0>  @MARK_TOP_ACCENTS;
```

NOTE! All mark class definition statements must precede any use of a mark class in the feature file. Once any position statement has referenced a mark class, no more mark statements are allowed.

NOTE! The mark classes used within a single lookup must be disjoint: none may include a glyph which is in another mark class that is used within the same lookup.

NOTE! If a GDEF table is not explicitly defined in the feature file, then an implementation of this syntax will create one. In this case, it will use the set of defined mark classes to define the mark glyphs for the GDEF GlyphClass. In this case, all mark glyph classes must be disjoint, and no base glyph may be used also as a mark glyph.

The set of mark classes have an implicit order. Any implementation should order the mark classes in the order of occurrence of the first definition statement for a mark glyph class. In the example:

```
markClass [acute grave] <anchor 350 0>  @MARK_TOP_ACCENTS;
markClass [cedilla hook] <anchor  300 0>  @MARK_BOTTOM_ACCENTS;
markClass [dieresis umlaut] <anchor 400 0>  @MARK_TOP_ACCENTS;
```

the mark class order will be:

```
@MARK_TOP_ACCENTS # mark class index 0
@MARK_BOTTOM_ACCENTS # mark class index 1
```

4.g. subtable

The feature file implementation must insert subtable breaks among the rules for a particular lookup if needed. For example, if a set of alternate substitution rules specified in the feature file exceeds the subtable size limit, several subtables must be automatically created.

The "subtable" statement may be used as follows:

```
subtable;
```

to explicitly force a subtable break after the previous rule.

[This is currently supported only for Pair Adjustment Positioning Format 2 (i.e. pair class kerning) in order to reduce subtable size. See §6.b.iii for details.]

4.h. Examples

Example 1. The following is an example of an entire feature file and demonstrates the two ways to register features under language systems (see §4.b above):

```
languagesystem DFLT dflt;
languagesystem latn dflt;
languagesystem latn DEU;
languagesystem latn TRK;
languagesystem cyrl dflt;

feature smcp {
  sub [a-z] by [A.sc-Z.sc];
  # Since all the rules in this feature are of the same type, they will be grouped in a single lookup.
  # Since no script or language keyword has been specified yet,
  # the lookup will be registered for this feature under all the language systems.
} smcp;

feature liga {
      sub f f by f_f;
      sub f i by f_i;
      sub f l by f_l;
  # Since all the rules in this feature are of the same type, they will be grouped in a single lookup.
  # Since no script or language keyword has been specified yet,
  # the lookup will be registered for this feature under all the language systems.

  script latn;
    language dflt;
```

```
        # lookupflag 0;     (implicit)
        sub c t by c_t;
        sub c s by c_s;
        # The rules above will be placed in a lookup that is registered for all the
        # specified languages for the script latn, but not any other scripts.

      language DEU;
        # script latn;      (stays the same)
        # lookupflag 0;     (stays the same)
        sub c h by c_h;
        sub c k by c_k;
        # The rules above will be placed in a lookup that is registered only under the
        # script latn, language DEU.

      language TRK;
        # This will inherit both the top level default rules - the rules
        # defined before the first 'script' statement, and the script-level default rules for 'latn':
        # all the lookups of this feature defined after the 'script latn' statement, and before the language DEU statement.
        # If TRK were not named here, it would not inherit the default rules for the script latn.
  } liga;

  feature kern {
    pos a y -150;
    # [more pos statements]
    # All the rules in this feature will be grouped in a single lookup
    # that is is registered under all the language-systems.
  } kern;
```

In the above example feature file, the 'smcp' and 'kern' features will be registered under the 'DFLT'/'dflt', latn'/'dflt', 'latn'/'DEU ', , 'latn'/'TRK' and 'cyrl'/'dflt' language systems since no explicit "script" or "language" statements are present in those features.

In the 'liga' feature, the f_f, f_i and f_l ligature substitutions will be applied under all all language systems. The c_t and c_s ligature substitutions will be applied under all languages of the script latn, but not under any other scripts. The c_h and c_k ligature substitutions will be applied when the language is German (i.e. they are registered only under 'latn'/'DEU ').

Example 2. The following example illustrates labeled lookup blocks and the use of the exclude_dflt keyword:

```
  languagesystem DFLT dflt;
  languagesystem latn dflt;
  languagesystem latn DEU;
  languagesystem cyrl dflt;
  languagesystem cyrl SRB;
  languagesystem grek dflt;

  feature liga {
         # start of default rules that are applied under all language systems.
        lookup HAS_I {
         sub f f i by f_f_i;
         sub f i by f_i;
        } HAS_I;

        lookup NO_I {
         sub f f l by f_f_l;
         sub f f by f_f;
        } NO_I;

# end of default rules that are applied under all language systems.

     script latn;
        language dflt;
# default lookup for latn included under all languages for the latn script

        sub f l by f_l;
        language DEU;
# default lookups included under the DEU language..
        sub s s by germandbls;   # This is also included.
        language TRK exclude_dflt;   # default lookups are excluded.
         lookup NO_I;            #Only this lookup is included under the TRK language

     script cyrl;
       language SRB;
         sub c t by c_t; # this rule will apply only under script cyrl language SRB.
  } liga;
```

The 'DFLT'/'dflt', 'cyrl'/'dflt', and 'grek'/'dflt' language systems will all contain the default rules for 'liga' functionality, the

rules in the lookups HAS_I and NO_I.

Under the 'latn' script, all languages but TRK will contain the default lookups, and the f_l ligature rule.

The germandbls ligature will apply only for the 'latn'/'DEU ' language system.

The 'latn'/'TRK ' languagesystem will contain only the NO_I lookup.

The 'cyrl'/'SRB ' languagesystem will contain all the default rules, and the c_t ligature rule.

Note that if you specify no explicit rules or lookup references after a script and language statement, that the effect is to include all the default rules for all scripts for the feature. Note also that lookup HAS_I must be placed before lookup NO_I since the f_f_i substitution must precede the f_f substitution when both are applied. (See §7, "Ordering of lookups and rules in the feature file," below).

The ordering of ligature rules within a particular lookup does not matter, excepting contextual rules, as the implementation will sort non-contextual rules in order to avoid conflict. For example, in lookup HAS_I, the f_i substitution may be placed before the f_f_i substitution, because the implementation will sort the f_f_i substitution first when writing the lookup to the font. (See §5.d, "Ligature substitution," below).

5. Glyph substitution (GSUB) rules

Glyph substitution rules begin with the keyword "substitute"; this keyword may be abbreviated as "sub". (The "ignore" keyword may precede the "substitute" keyword in some cases.) The GSUB lookup type is auto-detected from the format of the rest of the rule.

5.a. [GSUB LookupType 1] Single substitution

A Single Sub rule is specified in one of the following formats:

```
substitute <glyph> by <glyph>;           # format A
substitute <glyphclass> by <glyph>;      # format B
substitute <glyphclass> by <glyphclass>; # format C
```

Format B specifies that any glyph in the target glyph class must be replaced by the same replacement glyph.

Format C specifies that any of the glyphs in the target glyph class must be replaced by its corresponding glyph (in the order of glyphs in the glyph classes) in the replacement glyph class. If the replacement is a singleton glyph class, then the rule will be treated identically to a format B rule. If the replacement class has more than one glyph, then the number of elements in the target and replacement glyph classes must be the same.

For example:

```
sub a by A.sc;                                  # format A
substitute [one.fitted one.oldstyle one.tab.oldstyle] by one; # format B
substitute [a - z] by [A.sc - Z.sc];            # format C
substitute @Capitals by @CapSwashes;            # format C
```

The third line in the above example produces an identical representation in the font as:

```
substitute a by A.sc;
substitute b by B.sc;
substitute c by C.sc;
# ...
substitute z by Z.sc;
```

5.b. [GSUB LookupType 2] Multiple substitution

A Multiple Sub rule is specified as:

```
substitute <glyph> by <glyph sequence>;
```

<glyph sequence> contains two or more glyphs. It may not contain glyph classes. (If it did, the rule would be ambiguous as to which replacement sequence were required.) For example:

```
substitute f_f_i by f f i;          # Ligature decomposition
```

5.c. [GSUB LookupType 3] Alternate substitution

An Alternate Sub rule is specified as:

```
substitute <glyph> from <glyphclass>;
```

For example:

```
substitute ampersand from [ampersand.1 ampersand.2 ampersand.3];
```

5.d. [GSUB LookupType 4] Ligature substitution

A Ligature Sub rule is specified as:

```
substitute <glyph sequence> by <glyph>;
```

<glyph sequence> must contain two or more of <glyph|glyphclass>. For example:

```
substitute [one one.oldstyle] [slash fraction] [two two.oldstyle] by onehalf;
```

Since the OpenType specification does not allow ligature substitutions to be specified on target sequences that contain glyph classes, the implementation software will enumerate all specific glyph sequences if glyph classes are detected in <glyph sequence>. Thus, the above example produces an identical representation in the font as if all the sequences were manually enumerated by the font editor:

```
substitute one       slash   two       by onehalf;
substitute one.oldstyle slash    two       by onehalf;
substitute one       fraction two       by onehalf;
substitute one.oldstyle fraction two       by onehalf;
substitute one       slash    two.oldstyle by onehalf;
substitute one.oldstyle slash    two.oldstyle by onehalf;
substitute one       fraction two.oldstyle by onehalf;
substitute one.oldstyle fraction two.oldstyle by onehalf;
```

A contiguous set of ligature rules does not need to be ordered in any particular way by the font editor; the implementation software must do the appropriate sorting. So:

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

will produce an identical representation in the font as:

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

5.e. [GSUB LookupType 5] Contextual substitution

This LookupType is a functional subset of GSUB LookupType 6, chaining contextual substitution. Thus, all desired rules of this LookupType can be expressed in terms of chaining contextual substitution rules.

5.f. [GSUB LookupType 6] Chaining contextual substitution

5.f.i. Specifying a Chain Sub rule and marking sub-runs

A Chain Substitution rule target sequence has three parts: backtrack, input, and lookahead glyph sequences. A glyph sequence comprises one or more glyphs or glyph classes.

The most important is input glyph sequence. This is the sequence of glyphs and glyph classes to which substitution operations are applied. Optionally, a prefix (also known as backtrack) glyph sequence may be specified, as well as a suffix (also known as lookahead) glyph sequence. The entire sequence of glyphs -- prefix plus input plus suffix -- must match in the current context for the rule to be applied. The match sequence is aligned to the current context by aligning the first glyph of the input sequence with the current glyph of the text being processed. If the rule is matched, then the current context moves the current glyph pointer ahead in the original text by the length of the input sequence. Note that in the FDK syntax, the entire context string (backtrack sequence + input sequence + look-ahead sequence) are all written in the text string order. This is worth emphasis, as inside the lookup rule, the glyphs of the backtrack sequence are written in reverse order from the text to be matched. Developers of font editing tools who know this are sometimes confused by the FDK syntax.

For each glyph or glyph class in the input sequence, the contextual rule may specify one lookup (§4.e) to be applied at that position. Note that the specified lookup may contain many rules; the implementation must ensure that only one rule in a referenced lookup will match at that position in the input sequence. Lookups cannot be specified for the glyphs or glyph classes in the backtrack and lookahead sequences.

The input sequence is defined by appending the mark (') character to all the glyph names and class names within the input sequence.

The most general form of the contextual substitution rule is to explicitly reference named lookups in the rule.

Example 1. Define two stand-alone lookups (§4.e), and then reference them in the input sequence of contextual substitution rules with the keyword "lookup" and the lookup name.

```
lookup CNTXT_LIGS {
    substitute f i by f_i;
    substitute c t by c_t;
 } CNTXT_LIGS;
```

```
lookup CNTXT_SUB {
    substitute n by n.end;
    substitute s by s.end;
} CNTXT_SUB;

feature test {
    substitute [ a e i o u] f' lookup CNTXT_LIGS i' n' lookup CNTXT_SUB;
    substitute [ a e i o u] c' lookup CNTXT_LIGS t' s' lookup CNTXT_SUB;
} test;
```

Note that both the contextual substitution rules use the same lookups. This is because there is more than one rule in each referenced lookup, and different rules within the referenced lookups will match in the different contexts. In the first contextual substitution rule, the lookup `CNTXT_LIGS` will be applied at the input sequence glyph "f", and the glyphs "f" and "i" will be replaced by "f_i". The lookup `CNTXT_SUB` will be applied at the input sequence glyph "n", and the glyph "n" will be replaced by "n.end". This will happen only when the sequence "f i n" is preceded by any one of the glyphs "a e i o u". Likewise, in the second contextual substitution rule the glyphs "c" and "t" will be replaced by "c_t", and the glyph "s" will be replaced by "s.end". This will happen only when the sequence "c t s" is preceded by any one of the glyphs "a e i o u".

This form of the contextual substitution rule is the most flexible. You can specify a substitution lookup for more than one input sequence glyph or glyph class, the referenced lookups can be of different types, and the referenced lookups can have different lookup flags that the parent contextual lookup. The drawback is that it is difficult to understand what substitution rule will be applied, and the implementation may not warn if the referenced lookup does not contain a rule that matches the context.

If there is only a single substitution operation, and it is either single substitution or ligature substitution, then the operation can be specified in-line and its type will be auto-detected from the input and replacement sequence in the same way as in their corresponding stand-alone (i.e. non-contextual) statements.

Example 2. This calls a Single Sub rule. The rule below means: in sequences "a d" or "e d" or "n d", substitute "d" by "d.alt".

```
substitute [a e n] d' by d.alt;
```

This format requires that there be only one glyph of glyph class in the input sequence.

Example 3. This also calls a Single Sub rule. The rule below means: if a capital letter is followed by a small capital, then replace the small capital by its corresponding lowercase letter.

```
substitute [A-Z] [A.sc-Z.sc]' by [a-z];
```

This format requires that there be only one glyph of glyph class in the input sequence.

Example 4. This calls a Ligature Sub lookup. The rule below means: in sequences "e t c" or "e.begin t c", substitute the first two glyphs by the ampersand.

```
substitute [e e.begin]' t' c by ampersand;
```

This format will assume that entire input sequence is the sequence of ligature components.

5.f.ii. Specifying exceptions to the Chain Sub rule

Exceptions to a chaining contextual substitution rule are expressed by inserting a statement of the following form anywhere before the chaining contextual rule and in the same lookup as it:

```
ignore substitute <backtrack glyph sequence>*
        <marked glyph sequence>
        <lookahead glyph sequence>*;
```

The backtrack and lookahead sequences may be omitted, but there must be at least one marked glyph or glyph class.

The "ignore substitute" statement works by creating subtables in the GSUB that tell the OT layout engine simply to match the specified sequences, and not to perform any substitutions on them. As a result of the match, remaining rules (i.e. subtables) in the lookup will be skipped when the rule matches. (See the OT layout algorithm in §7.a.)

Example 1. Ignoring specific sequences: The "ignore substitute" rules below will block any subsequent rules that specifies a substitution for "d" when the context around "d" matches any of the sequences "f a d", "f e d", or "a d d".

Note that the marked glyphs in the exception sequences indicate where a substitution would have occurred; this is necessary for the OpenType layout engine to correctly handle skipping this sequence.

```
ignore substitute f [a e] d'
ignore substitute a d' d;
substitute [a e n] d' by d.alt;
```

Example 2. Matching a beginning-of-word boundary:

```
ignore substitute @LETTER f' i';
substitute f' i' by f_i.begin;
```

The example above shows how a ligature may be substituted at a word boundary. @LETTER must be defined to include all glyphs considered to be part of a word. The substitute statement will get applied only if the sequence doesn't match "@LETTER f i"; i.e. only at the beginning of a word.

Example 3. Matching a whole word boundary:

```
ignore substitute @LETTER a' n' d', a' n' d' @LETTER;
substitute a' n' d' by a_n_d;
```

In this example, the a_n_d ligature will apply only if the sequence "a n d" is neither preceded nor succeeded by a @LETTER.

Example 4. This shows a specification for the contextual swashes feature:

```
feature cswh {

  # --- Glyph classes used in this feature:
  @BEGINNINGS = [A-N P-Z Th m];
  @BEGINNINGS_SWASH = [A.swash-N.swash P.swash-Z.swash T_h.swash m.begin];
  @ENDINGS = [a e z];
  @ENDINGS_SWASH = [a.end e.end z.end];

  # --- Beginning-of-word swashes:
  ignore substitute @LETTER @BEGINNINGS';
  substitute @BEGINNINGS' by @BEGINNINGS_SWASH;

  # --- End-of-word swashes:
  ignore substitute @ENDINGS' @LETTER;
  substitute @ENDINGS' by @ENDINGS_SWASH;

} cswh;
```

If a feature only targets glyphs at the beginning or ending of a word, such as the 'init' and 'fina' features, then the application could be made responsible for detecting the word boundary; the feature itself would be simply defined as the appropriate substitutions without regard for word boundary. Such application responsibilities must be described in the feature tag registry.

5.g. [GSUB LookupType 7] Extension substitution

The "useExtension" keyword specifies creating lookups of this lookup type. See §4.e and §8.a.

5.h. [GSUB LookupType 8] Reverse Chaining Single Substitution

A Reverse Chaining Single Substitution shares the same syntax as a Chaining Contextual Substitution rule. The syntactic differences are that it can specify only single substitutions, the marked target sequence can consist of only a single glyph or glyph class, and the rule is specified with the keyword **reversesub** or **rsub**. An application's layout engine will also treat this rule differently than any other rule type; the lookup is applied to the text string in the reverse of the logical reading order.

The rule is specified as follows:

```
reversesub [a e n] d' by d.alt;
```

6. Glyph positioning (GPOS) rules

Glyph positioning rules begin with the keyword "position"; this keyword may be abbreviated as "pos". (The "enumerate" or "ignore" keywords may precede the "position" keyword in some cases.) The GPOS lookup type is auto-detected from the format of the rest of the rule.

Glyph positioning is specified in terms of metrics [§2.e.ii], device tables [§2.e.iii], value records [§2.e.iv], and anchors [§2.e.vii]. In all positioning rules, these are inserted immediately after the glyph(s) they apply to, with the exception of Pair Pos format B.

6.a. [GPOS LookupType 1] Single adjustment positioning

A Single Pos rule is specified as:

```
position <glyph|glyphclass>
<valuerecord>;
```

Here, the <glyph|glyphclass> is adjusted by the <valuerecord> [§2.e.iv]. For example, to reduce the left and right side-bearings of a glyph each by 80 design units:

```
position one <-80 0 -160 0>;
```

6.b. [GPOS LookupType 2] Pair adjustment positioning

6.b.i. Specific and class pair kerning

Rules for this LookupType are usually used for kerning, and may be in either of 2 formats:

- Pair Pos format A: *[Currently not implemented.]*

  position <glyph|glyphclass> <valuerecord>

  <glyph|glyphclass> <valuerecord>;

  The first <valuerecord> [§2.e.iv] corresponds to the first <glyph|glyphclass>, and the second <valuerecord> corresponds to the second <glyph|glyphclass>. The following example illustrates an unusual way to specify a kern value of -100:

  position T <-60> a <-40 0 -40 0>;

- Pair Pos format B:

  position <glyph|glyphclass> <glyph|glyphclass>
  <valuerecord>; # for first <glyph|glyphclass>

This format is provided since it closely parallels the way kerning is expressed in AFM files. Thus, it is a shorter way of expressing:

position <glyph|glyphclass> <valuerecord format A> <glyph|glyphclass>
<**NULL**>;

Kerning can most easily be expressed with this format. This will result in adjusting the first glyph's×advance, except when in the 'vrkn' feature, in which case it will adjust the first glyph's Y advance. Some examples:

pos T a -100; # specific pair (no glyph class present)
pos [T] a -100; # class pair (singleton glyph class present)
pos T @a -100; # class pair (glyph class present, even if singleton)
pos @T [a o u] -80; # class pair

Note that in both formats A and B, if at least one glyph class is present (even if it is a singleton glyph class), then the rule is interpreted as a class pair; otherwise, the rule is interpreted as a specific pair.

In the 'kern' feature, the specific glyph pairs will typically precede the glyph class pairs in the feature file, mirroring the way that they will be stored in the font. (See §7, "Ordering of lookups and rules in the feature file," below.)

feature kern {
# specific pairs for all scripts
# class pairs for all scripts
} kern;

6.b.ii. Enumerating pairs

If some specific pairs are more conveniently represented as a class pair, but the editor does not want the pairs to be in a class kerning subtable, then the class pair must be preceded by the keyword "enumerate" (which can be abbreviated as "enum"). The implementation software will enumerate such pairs as specific pairs. Thus, these pairs can be thought of as "class exceptions" to class pairs. For example:

@Y_LC = [y yacute ydieresis];
@SMALL_PUNC = [comma semicolon period];

enum pos @Y_LC semicolon -80; # specific pairs
pos f quoteright 30; # specific pair
pos @Y_LC @SMALL_PUNC -100; # class pair

The enum rule above can be replaced by:

pos y semicolon -80;
pos yacute semicolon -80;
pos ydieresis semicolon -80;

without changing the representation in the font.

6.b.iii. Subtable breaks

The implementation software will insert a subtable break within a run of class pair rules if a single subtable cannot be created due to class overlap. A warning will be emitted. For example:

pos [Ygrave] [colon semicolon] -55; # [line 99] In first subtable
pos [Y Yacute] period -50; # [line 100] In first subtable
pos [Y Yacute Ygrave] period -60; # [line 101] In second subtable

will produce a warning that a new subtable has been started at line 101, and that some kern pairs within this subtable may never be accessed. Note that this allows the font to be built, but the result will not match the developer's intention. The kerning feature will not work as expected until the causes for all such errors are removed. The pair (Ygrave, period) will have a value of 0 if the above example comprised the entire lookup, since Ygrave is in the coverage (i.e. union of the first glyphs) of the first subtable. One way to understand this is to imagine a lookup table of kern class pairs as a spreadsheet

of all possible pairs of kern left-side classes that are used in the lookup table with all the kern right-side classes that are used in the lookup table. Imagine each left side class is the title of a row in the spreadsheet, and each right side class is the title of a column. A glyph can be put in only one row title, and in only one column title. All glyphs not named in a row title get put together in a special row title. All glyphs not named in a column title get put together in a special column title. When you specify the value of a class pair, you are specifying the value in only one cell of the spreadsheet. When you specify a series of kern pair rules between a particular left side class and a series of right side classes, you are filling in a series of cells in the row for the specific left side class. All cells for which no values are specified are set to 0. When programs look for a kern value between "Ygrave" and something else, they look through the list of left side class definitions to find the first occurrence of 'Ygrave'. By definition, the first spreadsheet row which includes "Ygrave" will define the kern pair value of "Ygrave" with all other right-side classes, e.g spreadsheet columns. Since a pair value with a right-side period has not been explicitly defined at this point, the default value is 0. Since the programs will not look further than this row, the kernclass pair:

pos [Y Yacute Ygrave] period -60;

will never be used.

Sometimes the class kerning subtable may get too large. The editor can make it smaller by forcing subtable breaks at any point by inserting the statement:

subtable;

between two class kerning rules. The new subtable created will still be in the same lookup, so the editor must ensure that the coverages of the subtables thus created do not overlap, since the processing rules will not find and report a conflict.

When seeking to decrease the class table size, it is best to place subtable breaks between blocks of rules where there is no cross linking, such that no left side class in one block is used with any right side class in the other block. However, in most large Western fonts, such groups are so small that breaking them into separate subtables does not yield much decrease in the the overall lookup size. In this common case, an adequate strategy is to first divide the entire list of kern class rules in two roughly equal blocks with a subtable break. If this does not make the class kern tables small enough, then continue to subdivide each block of rules in two with a subtable break. Because the class definitions must be repeated for each subtable, a point of diminishing returns usually comes with around 6 subtable breaks.

6.c. [GPOS LookupType 3] Cursive attachment positioning

A Cursive Pos rule is specified as:

position cursive <glyph|glyphclass> <anchor> # Entry anchor
<anchor>; # Exit anchor

The first <anchor> [§2.e.vii] indicates the entry anchor point for <glyph|glyphclass>; the second, the exit anchor point.

For example, to define the entry point of glyph meem.medial to be at x=500, y=20, and the exit point to be at x=0, y=-20:

position cursive meem.medial <**anchor** 500 20> <**anchor** 0 -20>;

A glyph may have a defined entry point, exit point, or both. <anchor> format D, the null anchor, must be used to indicate that an <anchor> is not defined.

position cursive meem.end <**anchor** 500 20> <**anchor** NULL >;

6.d. [GPOS LookupType 4] Mark-to-Base attachment positioning

A Mark-to-Base Pos rule is specified as:

position base <glyph|glyphclass> # base glyph(s)
<anchor> **mark** <named mark glyphclass> + # anchor and mark glyph class; repeated for each attachment point on the base glyphs(s) name
;

Each <anchor> [§2.e.vii] indicates the anchor point on the base glyph(s) to which the mark class's anchor point should be attached.

A single Mark-To-Base statement must specify all the anchor points and their attaching mark classes.

This rule type does not actually support base glyph classes: the feature file syntax allows this in order to compactly specify Mark-To-Base rules for the set of glyphs which have the same anchor points. A feature file rule which uses a glyph class for the base glyph is expanded in the font to a separate rule for each glyph in the base class, although they will share the same anchor and mark class records.

The named mark glyph classes and the anchor points of all the mark glyphs in the named mark classes must have been previously defined in the feature file by **markClass** statements [§4.f].

NOTE! The mark classes used within a single lookup must be disjoint: none may include a glyph which is in another mark class that is used within the same lookup.

For example, to specify that the anchor of mark glyphs acute and grave is at x=30, y=600, and that the anchor of mark glyphs dieresis and umlaut is at x=60, y=600, and to position the anchor point of the four mark glyphs at anchor point x=250, y=450 of glyphs a, e, o and u:

```
markClass [acute grave] <anchor 150 -10> @TOP_MARKS;
markClass [dieresis umlaut] <anchor 300 -10> @TOP_MARKS;
markClass [cedilla] <anchor 300 600> @BOTTOM_MARKS;

position base [a e o u] <anchor 250 450> mark @TOP_MARKS
                <anchor 250 -10> mark @BOTTOM_MARKS;
```

All the base glyphs in a base glyph class must share the same anchor points for all mark classes, otherwise separate statements are needed.

```
position base [e o] <anchor 250 450> mark @TOP_MARKS
            <anchor 250 -12> mark @BOTTOM_MARKS;
position base [a u] <anchor 265 450> mark @TOP_MARKS
            <anchor 250-10> mark @BOTTOM_MARKS;
```

6.e. [GPOS LookupType 5] Mark-to-Ligature attachment positioning

A Mark-to-Ligature Pos rule is specified as:

```
position ligature <ligature glyph|glyphclass>   # ligature glyph or glyph class
        # anchor and named mark glyph class;
        # repeated for each anchor point on the first component glyph
        <anchor> mark <named mark glyph class> +
        # Start of anchor and mark info for the next ligature component.
        ligComponent

# anchor and named mark glyph class;

# repeated for each anchor point on the next component glyph

<anchor> mark <named mark glyph class>


# The block of ligComponent its anchor-mark classes

# is repeated for each ligature component.

;
```

The statement must specify all the anchor-mark class pairs for all the ligature components. It follows the form of the Mark-To-Base rule, except that a set of anchor-mark class pairs must be specified for each component glyph in the ligature. The set of anchor-mark class pairs for one component is separated for the set of the next component by the 'ligComponent' keyword. If there are no anchor points on a component, it must still specify at least one anchor, which should be the NULL anchor. It is not required that each component have the same number of anchor points.

The named mark glyph classes and the anchor points of all the mark glyphs in the named mark classes must have been previously defined in the feature file by **markClass** statements [§4.f].

The example in the OpenType specification for this LookupType could be expressed as:

```
# 1. Define mark anchors:
markClass sukun    <anchor 261 488> @TOP_MARKS;
markClass kasratan <anchor 346 -98> @BOTTOM_MARKS;

# 2. Define mark-to-ligature rules:
position ligature lam_meem_jeem
    <anchor 625 1800> mark @TOP_MARKS    # mark above lam
    ligComponent    # start specifying marks for meem
    <anchor 376 -368> mark @BOTTOM_MARKS    #mark below meem
    ligComponent    # start specifying marks for jeem
    <anchor NULL>    # jeem has no marks
    ;
```

Note that a NULL anchor needs to be specified for a ligature component only when it has no non-NULL anchors. Otherwise, the implementation will supply a NULL anchor for each mark class that is not used by a ligature component.

If a glyph class is used, each ligature in the glyph class must have the same number of components and the same anchor positions on each component.

6.f. [GPOS LookupType 6] Mark-to-Mark attachment positioning

A Mark-to-Mark Pos rule is specified as:

```
position mark <glyph|glyphclass> # base mark glyph(s)
    <anchor>  mark <named mark glyphclass> +    # anchor and mark glyph class; repeated for each attachment point on the base glyphs(s) name
    ;
```

This rule is distinguished from a Mark-to-Base Pos rule [§6.d] by the first "mark" keyword. Otherwise, it has the same syntax and restrictions.

The example in the OpenType specification for this LookupType could be expressed as:

```
# 1. Define name mark class:
markClass damma <anchor 189 -103> @MARK_CLASS_1;
# 2. Define mark-to-mark rule:
position mark hanza <anchor 221 301> mark @MARK_CLASS_1;
```

6.g. [GPOS LookupType 7] Contextual positioning

This LookupType is a functional subset of GPOS LookupType 8, chaining contextual positioning. Thus, all desired rules of this LookupType can be expressed in terms of chaining contextual positioning rules.

6.h. [GPOS LookupType 8] Chaining contextual positioning

Positioning rules in this lookup type are supported as of FDK 2.0

6.h.i. Specifying a Chain Positioning rule and marking sub-run

A Chain Positioning rule target sequence has three parts: backtrack, input, and lookahead glyph sequences. A glyph sequence comprises one or more glyphs or glyph classes.

The most important is input glyph sequence. This is the sequence of glyphs and glyph classes to which positioning operations are applied. Optionally, a prefix (also known as backtrack) glyph sequence may be specified, as well as a suffix (also known as lookahead) glyph sequence. The entire sequence of glyphs -- prefix plus input plus suffix -- must match in the current context for the rule to be applied. The match sequence is aligned to the current context by aligning the first glyph of the input sequence with the current glyph of the text being processed. If the rule is matched, then the current context moves the current glyph pointer ahead in the original text by the length of the input sequence. Note that in the FDK syntax, the entire context string (backtrack sequence + input sequence + look-ahead sequence) are all written in the text string order. This is worth emphasis, as inside the lookup rule, the glyphs of the backtrack sequence are written in reverse order from the text to be matched. Developers of font editing tools who know this are sometimes confused by the FDK syntax.

For each glyph or glyph class in the input sequence, the contextual rule may specify one lookup (§4.e) to be applied at that position. Note that the specified lookup may contain many rules; the implementation must ensure that only one rule in a referenced lookup will match at that position in the input sequence. Lookups cannot be specified for the glyphs or glyph classes in the backtrack and lookahead sequences.

The input sequence is defined by appending the mark (') character to all the glyph names and class names (and only these names) within the input sequence. Applying the mark (') character to keywords such as anchor and mark or a value record will result in a syntax error.

6.h.ii. Specifying Contextual Positioning with explicit lookup references

The most general form of the contextual substitution rule is to explicitly reference named lookups in the rule.

Example 1. Define two stand-alone lookups (§4.e), and then reference them in the input sequence of contextual positioning rules with the keyword "lookup" and the lookup name.

```
lookup CNTXT_PAIR_POS {
    position T o -10;
    position T c by -12;
} CNTXT_PAIR_POS;

lookup CNTXT_MARK_TO_BASE {
    position base o
    position base c
} CNTXT_MARK_TO_BASE;

feature test {
    substitute T' lookup CNTXT_PAIR_POS [o c]' @ALL_MARKS' lookup CNTXT_MARK_TO_BASE;
} test;
```

This rule has only an input sequence, and no backtrack or lookahead sequence. It will match when the current glyph is 'T', followed by either 'o' or 'c', followed by any mark glyph. The lookup CNTXT_PAIR_POS will applied to the 'T', and the lookup CNTXT_MARK_TO_BASE will be applied to the glyphs in the class @ALL_MARKS.

This form of the contextual positioning rule is the most flexible. You can specify a positioning lookup for more than one input sequence glyph or glyph class, the referenced lookups can be of different types, and the referenced lookups can have different lookup flags that the parent contextual lookup. The drawback is that it is difficult to understand what position rule will be applied, and the implementation may not warn if the referenced lookup does not contain a rule that matches the context.

When it is acceptable to specify a positioning rule for only one input glyph or glyphclass in the input sequence, and that the referenced lookup have the same lookup flag as the parent contextual lookup, then you can specify a contextual rule with the positioning rule in-line. This is much easier to understand.

6.h.iii. Specifying Contextual Positioning with in-line single positioning rules

Example 1:

```
position [quoteleft quotedblleft ][Y T]' <0 0 20 0 >  [quoteright quotedblright];
position [quoteleft quotedblleft ][Y T]' 20  [quoteright quotedblright];
```

Both of these rules have an input sequence of a single glyph position, for which the glyph class [Y T] is specified. The marked glyph class is followed by a value record. The first form shows a full value record which allows you to alter both the (x,y) coordinates of the origin and the (x,y) coordinates of the advance width. The second rule shows the simple form of the value record, which specifies a value for only a change to thexvalue of the advance width. Note that the value record modifies the glyph which it follows. These both increase the advance width of Y or T by 20, when preceded by either quoteleft or quotedblleft, and followed by quoteright or quotedblright. Note that not all marked glyphs or glyphclasses in the input sequence must be followed by a value record; if this is omitted, then the item's positioning info will not be affected.

Example 2:

```
position s f' 10 t;
position s f'10 t' -5 period;
```

The first example specifies a kern pair "ft" when preceded by "s", and increases the x-advance of f by 10. The second specifies a kern triplet "ft.", when preceded by "s". The x-advance of f is increased by 10, and the x-advance of t is decreased by 5. The entire run of marked glyphs will be consumed by a rule; in the first case, after matching this rule, the set of rules in current lookup will next be applied starting at the glyph "t". In the second case, the rules will next be applied starting at the glyph "period".

Special notes on contextual kerning

Contextual positioning rules must be in a different lookup than pair positioning rules, since the rules are of different lookup types. Because each lookup is applied independently of the other lookup(s) over the entire text stream, the positioning change specified in a pair kerning rule will be added to the positioning change specified in a contextual kerning rule, whenever the two rules match the same glyph pair in the text stream. This effect can be managed by specifying the contextual kerning rules values so that the sum of the pair positioning rule value and the contextual positioning rule value add to the desired value, as in example 3A.

Example 3A:

```
position L quoteright -150;
position quoteright A -120;
position L' 50 quoteright' 70 A;
```

Desired final kern adjustment: L' -100 quoteright' -50 A;

In this example, the intended kern correction for the triplet "L quoteright A" is an adjustment of -100 to the advance width of the L when followed by quoteright, and of -50 to the advance width of quoteright when followed by A. However, since the pair positioning rules will adjust the pair "L quoteright" by -150 and the pair "quoteright A" by -120, the adjustment values in the contextual rule for the triplet must be set as shown. This approach is feasible, but difficult to understand.

Another approach is to simply make all the kerning be contextual by marking the first glyph or glyph class of each pair positioning rule. Since all the kern rules will then be in a single lookup, only one rule will match in any context, and there is no need to figure out which rules add up. This solution is shown in example 3B using feature file syntax for contextual positioning. Notice, however, that the triplet rule had to defined before the other two rules. Otherwise, the pair positioning rules would have blocked the triplet's positioning adjustment.

Example 3B:

```
position L' -100 quoteright' -50 A;
position L' -150 quoteright;
position quoteright' -120 A;
position s f' 10 t period;
```

In order to make pair positioning rules easier to read and write as contextual kern pairs, the feature file syntax will identify a special case of contextual rule which contains only one marked glyph or glyph class, followed by one or more un-marked glyph or glyph class, plus a value record. This will be treated as a contextual pair positioning statement, and will be the only one case where a value record may follow an un-marked glyph. Example 3B can thus be written as example 3C. Both examples are exactly equivalent.

Example 3C:

```
position L' -100 quoteright' -50 A;
position L' quoteright -150; # special cases of contextual positioning
position quoteright' A -120; # where
value record follows un-marked glyph,

# making them exactly equivalent to 3B.
position s f' t 10 period;
```

Note that the following statement (Example 3D) is NOT a pair kerning statement, and would almost always be an error of intent.

Example 3D:

```
 position L' quoteright' -150;
```

This statement does two things that are not desirable in pair kerning statement. First, it decreases the advance width of quoteright, not L. Second, it will move the current glyph pointer forward by 2 glyphs, skipping over the quoteright so that quoteright will not be examined for matching kern rules.

The FDK syntax will not allow applying positioning lookups of different types in one contextual rule. For example, if you want to position sukun over lam_meem_jeem when followed by alef, and kern lam_meem_jeem with alef in this context, you need to put the mark and kern rules in different lookups.

Example 4:

```
 mark sukun <anchor 0 0>  @TOP_CLASS;

 lookup MARK_POS {
  position lam_meem_jeem' <anchor 625 1800>  mark @TOP_CLASS alef;
 } MARK_POS;

 lookup MARK_KERN {
  position lam_meem_jeem' 5 @TOP_CLASS
alef;
 } MARK_KERN;
```

The rule in lookup MARK_POS will position sukun over lam_meem_jeem when followed by alef. The second rule will add 5 to the advance width of lam_meem_jeem when followed by sukun and then by alef.

6.h.iv. Specifying Contextual Positioning with in-line cursive positioning rules

The contextual form of the cursive positioning rule consists of simply adding contextual glyphs or glyph classes before the 'cursive' keyword, and/or after the anchors. The base glyph must be marked as part of the input sequence; the others may or may not be marked.

```
 position @BACKTRACK_GLYPHS_FOR_MEEM cursive
meem.medial' <anchor 500 20> <anchor 0 -20> @LOOKAHEAD_GLYPHS_FOR_MEEM;
```

6.h.v. Specifying Contextual Positioning with in-line mark attachment positioning rules

For all three forms of the mark attachment rules - Mark-To-Base, Mark-To-Ligature, and Mark-To-Mark - the contextual form of the positioning rules consist of inserting glyph sequences in one or more of three places in the rule: 1) before the initial 'base | ligature | mark' keyword, 2) after the base glyph or glyph class, and 3) after all the anchor-mark class clauses.

At least one of the mark classes must be marked as part of the input sequence; the other glyphs or glyph classes in the contextual sequence may or may not be marked. There is special treatment of the mark classes. The implementation creates a glyph class which is the input glyph class to which the positioning lookup is applied. Each mark class that is marked as part of the input sequence is added to this glyph class. If the rule has four mark classes, and three are marked as part of the input sequence. the result is a single glyph class in the input sequence which contains the glyphs from the three marked mark classes. The base glyph or glyph class is also always added in the contextual sequence.

For example:

```
 position [T V F] base [a e o u] <anchor 250 450> mark @TOP_MARKS'
                    <anchor 250 -10> mark @BOTTOM_MARKS' @VOWELS;
```

This contextual rule will match when the current context matches the 4 item glyph sequence "[T V F] [a e o u] [top and bottom marks] @VOWELS". The input sequence has only one item, a glyph class which consists of all the glyphs from the two mark classes.

6.h.vi. Specifying exceptions to the Chain Pos rule

Exceptions to a chaining contextual positioning rule are expressed by inserting a statement of the following form anywhere before the chaining contextual rule and in the same lookup:

```
 ignore position <marked glyph sequence> (, <marked glyph sequence>)*;
```

This rule works in exactly the same was as specifying exceptions to a chaining contextual substitution rule [§5.f.ii].

6.i. [GPOS LookupType 9] Extension positioning

The "useExtension" keyword specifies creating lookups of this lookup type. See §4.e.

7. Ordering of lookups and rules in the feature file

7.a. An OpenType Layout engine's layout algorithm

The following is a reference summary of the algorithm used by an OpenType layout (OTL) engine to perform substitutions and positionings. The important aspect of this for a feature file editor is that each lookup corresponds to one "pass" over the glyph run (see step 4 below). Thus, each lookup has as input the accumulated result of all previous lookups in the LookupList (whether in the same feature or in other features).

1. All glyphs in the client's glyph run must belong to the same language system. (Glyph sequence matching may not occur across language systems.)

--- Do the following first for the GSUB and then for the GPOS: ---

2. Assemble all features (including any required feature) for the glyph run's language system.

3. Assemble all lookups in these features, in LookupList order, removing any duplicates. (All features and thus all lookups needn't be applied to every glyph in the run.)

4. For each lookup:

5. For each glyph in the glyph run:

6. If the lookup is applied to that glyph and the lookupflag doesn't indicate that that glyph is to be ignored:

7. For each subtable in the lookup:

8. If the subtable's target context is matched:

9. Do the glyph substitution or positioning,

--- OR: ---

If this is a (chain) contextual lookup do the following [(10)-(11)] in the subtable's Subst/PosLookupRecord order:

10. For each (sequenceIndex, lookupListIndex) pair:

11. Apply lookup[lookupListIndex] at input sequence[sequenceIndex] [steps (7)-(11)]

12. Goto the glyph after the input sequence matched in (8)
(i.e. skip any remaining subtables in the lookup).

The "target context" in step 8 above comprises the input sequence and any backtrack and lookahead sequences.

The input sequence must be matched entirely within the lookup's "application range" at that glyph (that contiguous subrun of glyphs including and around the current glyph on which the lookup is applied). There is no such restriction on the backtrack and lookahead sequences.

"Matching" includes matching any glyphs designated to be skipped in the lookup's LookupFlag.

7.b. Ordering of lookups and subtables

A lookup in the OpenType font will be created from each named lookup block [§4.e] or each run of rules with the same feature, script, language, lookupflag and lookup type attribute.

Lookups will be created in the GSUB/GPOS table's LookupList in the same order as the corresponding named lookup blocks or runs of rules in the feature file, except for the lookups that comprise the 'aalt' feature. These will always be created before all other features [§8.a].

A lookup may contain one or more subtables. Subtable breaks may have been inserted by the implementation software due to format restrictions, or they may have been explicitly requested by the editor [§4.f]. In either case, subtables will be created in the same order as the corresponding subtables in the feature file, if the order is relevant to OT layout. If the order is irrelevant, the implementation may choose to order subtables within a lookup in any manner.

Note that the lookup sharing mechanism (i.e. a "lookup" reference statement that refers to a named lookup block) is implemented simply by referring to the LookupList index of the lookup as many times as needed in the Feature tables.

7.c. Ordering of rules within a lookup

In the feature file, the ordering of rules within a lookup is important only for chaining contextual substitution and chaining contextual positioning rules. This is because in all other cases of LookupTypes (including ligature substitutions; see section 5.d), the appropriate ordering is automatically deduced, and the implementation sorts the rules accordingly when writing them to the font file.

8. Specially handled features

8.a. The all alternates ('aalt') feature

The aalt feature consists of a feature definition block which contains a series of statements in the form:

feature <feature tag>;

followed by one or more single and alternates substitution rules.

The feature file parser should create the 'aalt' feature from the feature file definition as follows:

1. Considering only features indicated by:
   feature <feature tag>;

   in the 'aalt' specification feature block (see example below), combine all single and alternate substitutions in those features (including single substitutions that appear within a chaining contextual rule) into groups with the first glyph in the group being the target glyph of the substitution. Subsequent elements of the group will be ordered by the order of the relevant rule in the feature file. Duplicate glyphs will be removed.

   The 'aalt' feature block must appear before the feature block of any <feature tag> it references in the above manner. It will also always be created as the first feature in the font (i.e. its lookups will be at the beginning of the GSUB LookupList).
2. Add any additional single and alternate substitutions in the 'aalt' specification to the groups that were created algorithmically, by step (i). This facility is provided to fine-tune the semantic groups, for instance, if certain glyphs weren't referenced in any of the features indicated in (i) above. This can also be used to override substitutions specified by including other features: for any target glyph, the alternate glyphs specified by this mechanism precede in order any other alternate glyphs.
3. If there are only two glyphs in a group, create a single substitution in the 'aalt' feature, with the first glyph being the target glyph and the second glyph being the replacement glyph. If there are more than two glyphs in a group, create an alternate substitution in the 'aalt' feature, with the first glyph being the target glyph and the remaining glyphs being the alternate set. These alternate glyphs will be sorted in the order that the source features are named in the 'aalt' definition, *not* the order of the feature definitions in the file. Alternates defined explicitly, as in step (ii) above, will precede all others.

The "useExtension" keyword:

The "useExtension" keyword may optionally precede "{" in the feature block syntax. The 'aalt' lookups will be created with the GSUB Extension lookup type if and only if the "useExtension" keyword is used. Note that since the Extension lookup types were added in OpenType specification v1.3, they will not be recognized by all OpenType layout parsers.

Specifying language system:

This feature will be registered under all language systems specified by "languagesystem" statements; see §4.b.i above.

The following are not allowed in the 'aalt' feature definition: "script", "language", "lookupflag", and "subtable" statements; named lookup blocks and lookup reference statements. The 'aalt' lookups will be created with LookupFlag 0.

Examples:

```
languagesystem DFLT dflt;
languagesystem latn dflt;
languagesystem latn TRK;
languagesystem cyrl dflt;

feature aalt {
  feature salt;
  feature smcp;
  substitute d by d.alt;
} aalt;

feature smcp {
  sub [a-c] by [A.sc-C.sc];
  sub f i by f_i;    # not considered for aalt
} smcp;

feature salt {
  sub a from [a.alt1 a.alt2 a.alt2];
  sub e [c d e] f by [c.mid d.mid e.mid];
  sub b by b.alt;
} salt;
```

The 'aalt' lookups from the above example will be registered under the default language systems of the 'DFLT', 'latn' and 'cyrl' scripts, and also under the 'latn'/'TRK ' language systems. The 'aalt' created would be the same as if the font editor had specified:

```
feature aalt {
  sub a from [a.alt1 a.alt2 a.alt3 A.sc];
  sub b from [b.alt B.sc];
  sub c from [c.mid C.sc];
  sub d from [d.alt d.mid];
  sub e by e.mid;
} aalt;
```

The following example will result in the 'aalt' lookups being created with the GSUB Extension lookup type:

```
feature aalt useExtension {
```

```
  feature salt;
  feature smcp;
  substitute d by d.alt;
  # ... other rules
 } aalt;
```

8.b. The optical size ('size') feature

This feature is unique in that it contains no substitution or positioning rules (the LookupCount field in its Feature table will always be 0).

The feature's data is accessed instead through the FeatureParams value of its Feature table.

Thus, the syntax for this feature is different from all other features. The feature block must contain:

- one "parameters" statement
- and zero or more "sizemenuname" statements.

No other feature file statements, blocks or keywords are permitted. (Comments are allowed.)

This feature will be created in the GPOS table and will be registered under all language systems specified by "languagesystem" statements (see §4.b.i above).

For example:

```
 feature size {
  parameters 100 # design size (decipoints)
             3  # subfamily identifier
            80  # range start (exclusive, decipoints)
           139; # range end (inclusive, decipoints)
  sizemenuname "Win MinionPro Size Name";
  sizemenuname 1 "Mac MinionPro Size Name";
  sizemenuname 1 21 0 "Mac MinionPro Size Name";
 } size;
```

See the OpenType feature tag registry for a description of the "parameters" statement fields. " decipoints" is a unit of 1/10 of a point.

These values may also be specified more directly as decimal point values, but a decimal point and following value is then required. For example, "8.0" and "80" will both result in the same value being stored in the font.

The parameter "sizemenuname" provides the menu name to be used for a group of fonts with the same subfamily identifier.

If the font is part of such a group, then the "sizemenuname" statement must be provided in order for the members of the group to be grouped together in a sub-menu under the specified menu name.

In the this case, we strongly recommend providing at least the two entries for Windows and Macintosh platform Roman script name strings. You may also include as any another localized name strings that may be useful.

If the font is not part of such a group, then the "sizemenuname" statement must be omitted, and all fields but the first (design size) for the parameter statement must be set to 0.

This can be used to indicate the intended design size for a font, even when it is not part of an optical size group.

The syntax of the "sizemenuname" statement follows that of the name table name strings, as described in §9.e.

The names specified by the "sizemenuname" statement are actually stored in the name table, with name ID's starting at the first unused name ID at or after 256.

8.c. Descriptive names for Stylistic Set ('ss01 - ss20') features

As of the OpenType specification 1.6, descriptive names are allowed for stylistic substitution features. These names are specified within a feature block for a Stylistic Set feature. The implementation will store the name strings in the name table, and will create a feature parameter data block which references them.

A single Stylistic Set feature block may contain more than one descriptive name in order to support different languages. These names are defined within a "featureNames" block that must be inside the stylistic set feature block, and must precede any of the rules in the feature. The syntax for a "featureNames" block is:

```
 featureNames {
   name < platform ID > < script ID > < language ID > < text string > ;
# This name entry is repeated for every script and language that you want to support.
 } ;
```

The syntax for the names is similar to that of the name table nameID entries (see see §9.e) - the only difference is that the introductory keyword is 'name', and the name ID value is omitted.

Example:

```
feature ss01 {
  featureNames {
      name "Feature description for MS Platform, script Unicode, language English";
# With no platform ID, script ID, or language ID specified, the implementation assumes (3,1,0×409).
      name 3 1 0×411 "Feature description for MS Platform, script Unicode, language Japanese";
      name 1 "Feature description for Apple Platform, script Roman, language unspecified";
# With only the platform ID specified, the implementation assumes script and language = Latin. For Apple this is (1,0,0).
      name 1 1 12 "Feature description for Apple Platform, script Japanese, language Japanese";    } ;
# --- rules for this feature ---
 } ss01;
```

9. Specifying or overriding table values

In addition to GSUB and GPOS OpenType layout features, the feature file provides for specifying or overriding values in certain other tables. These are specified within the corresponding table block:

```
table <table tag> {
  # ...
} <table tag>;
```

The following table values are currently supported:

9.a. BASE table

If no BASE table entry is specified in the feature file, no BASE table is created in the OpenType font.

```
table BASE {
  HorizAxis.BaseTagList <baseline tag>+;
  HorizAxis.BaseScriptList <script record> (, <script record>)*;
  HorizAxis.MinMax <minmax>;

  VertAxis.BaseTagList <baseline tag>+;
  VertAxis.BaseScriptList <script record> (, <script record>)*;
  VertAxis.MinMax <minmax>;
} BASE;
```

A <script record> is of the form:

```
<script tag> <default baseline tag> <base coord>+
```

<base coord> can take several formats: *[Currently only format A is implemented]*

```
<number>
# format A
 <number> <glyph>
<number>
# format B
 <number>
<device>
# format C
```

The baseline tags for each BaseTagList must be sorted in increasing ASCII order.

The number of baseline values for a particular script should be the same as the same as the number of baseline tags in the corresponding BaseTagList.

A <minmax> *[currently not implemented]* is of the form:

```
<script tag> <language tag>   # Defines the language system
    <base coord>,          # Min value for this language system
    <base coord>           # Max value for this language system
    [, <feature tag>       # (Optional) feature tag
    <base coord>,          # Min value for this feature tag
    <base coord>]          # Max value for this feature tag
    ;
```

For example:

```
table BASE {
  HorizAxis.BaseTagList              ideo romn;
  HorizAxis.BaseScriptList latn  romn  -120   0, cyrl  romn  -120   0,
                    grek  romn  -120   0, hani  ideo  -120   0,
                    kana  ideo  -120   0, hang  ideo  -120   0;
} BASE;
```

9.b. GDEF table

```
table GDEF {
   GlyphClassDef <glyphclass>*,   # base glyphs
                 <glyphclass>*,   # ligature glyphs
                 <glyphclass>*,   # mark glyphs
                 <glyphclass>;   # component glyphs
    Attach       <glyph|glyphclass> <number>+; # <number> is a contour point index

   LigatureCaret
   LigatureCaretByPos <glyph|glyphclass> <caret position value>+;
   LigatureCaretByIndex <glyph|glyphclass> <caret contour point index value>+;
 } GDEF:
```

The number of <caret value>s specified for a LigatureCaret must be: (number of ligature components) - 1.

Here is an example of a GDEF table block:

```
 table GDEF {
   GlyphClassDef @BASE, @LIGATURES, @MARKS, @COMPONENT;
   Attach noon.final 5;
   Attach noon.initial 4;
   LigatureCaretByPos f_i 400 380;
   LigatureCaretByPos [c_t c_s]  500;
   LigatureCaretByIndex f_f_i 23 46;
} GDEF;
```

The four class names in the GlyphClassDef are separated by commas. If you omit a class definition, you must still indicate its position with a comma. By the OpenType spec, any glyph not included in one of the class definition will be assigned glyph class index 0, and will not be included in any of the GlyphClass classes.

The MarkAttach classes of the GDEF table may not be specified explicitly in feature file syntax. They are instead created by the implementation from use of the "lookupflag MarkAttachmentType <class name>" statements. The class names may be from either regular classes definitions or mark class defintions.

The MarkGlyphSets classes of the GDEF table may not be specified explicitly in feature file syntax. They are instead created by the implementation from use of the "lookupflag UseMarkFilteringSet <class name>" statements. The class names may be from either regular classes definitions or mark class defintions.

If any mark class has been defined, or if any of the lookup flags for skipping glyphs of a certain class have been seen, the implementation will check if the GDEF keywords for defining the GlyphClassDefinition has been seen. If not, the implementation will fill them from the substitution and positioning rules, and will create a GDEF table even if there is no GDEF definition in the feature file. The LIGATURE and COMPONENT classes well be filled from the glyphs used in any GSUB Lookup Type 4, Ligature Substitution. The Marks class will be filled from all the glyphs in any of the mark classes used in positioning rules.

9.c. head table

The head table FontRevision value is used as the overall font version number, and should be incremented whenever any data in the fonts is changed. It is both specified and reported as a decimal number with three significant decimal places. The actual value stored in the font will, however, be a Fixed number (16.16 bit format). Due to the limited precision of this format, the value stored may differ by a small decimal fraction from that specified, but will always round to the same value when rounded to three fractional decimal places.

This value is also used as the source for the font version string in the name table name string ID 5 "Version".

```
 table head {
   FontRevision <fixed point number with three fractional decimal places>;
 } head;
```

Example 1:

```
 table head {
   FontRevision 1.1;
 } head;
```

This format is supported, but will cause a warning that the specification will be converted to "1.100". It will be stored in the font as 0×0001199A. A more exact decimal representation would be 1.10000610352, but it will be reported as "1.100".

Example 2:

```
 table head {
   FontRevision 1.001;
 } head;
```

This value be stored in the font as 0×00010042. A more exact decimal representation is 1.001007, but it will be reported as "1.001".

Example 3:

```
table head {
  FontRevision 1.500;
} head;
```

This value be stored in the font as 0x00018000, and will be reported as "1.500". The decimal and Fixed values are equal in this case.

9.d. hhea table

```
table hhea {
  CaretOffset <metric>;
  Ascender <metric>;
  Descender <metric>;
  LineGap <metric>;
} hhea;
```

For example:

```
table hhea {
  CaretOffset -50;
  Ascender 800;
  Descender 200;
  LineGap 200;
} hhea;
```

9.e. name table

```
table name {
  # name records
} name;
```

A name record is of the form:

```
nameid <id> [<string attribute>] <string>;
```

An <id> is a number specifying the id of the name string to be added to the name table. This number must be in the registered id range 0, 7-255. Note that ids 1-6 (Family, Subfamily, Unique, Full, Version, and FontName) are reserved by the implementation and cannot be overridden; doing so will elicit a warning message and the record will be ignored.

An optional <string attribute> is one or three space delimited numbers that specify the platform, platform-specific, and language ids to be stored in the name record of the name table. If only one number is specified it represents the platform id. The platform id may be either 1 or 3, corresponding to the Macintosh or Microsoft (hereafter called Windows) platforms, respectively. The other id numbers must be in the range 0-65535 but are not otherwise validated.

Decimal numbers must begin with a non-0 digit, octal numbers with a 0 digit, and hexadecimal numbers with a 0x prefix to numbers and hexadecimal letters a-f or A-F.

If some or all of the string attribute id numbers aren't specified their values are defaulted as follows:

```
 platform id     3
(Windows)
```

Windows platform selected:

```
 platspec id     1
(Unicode)
 language id
0x0409 (Windows default English)
```

Macintosh platform selected:

```
 platspec id     0
(Roman)
 language id     0
(English)
```

Putting this all together gives the following valid nameid formats and the ids that are assigned.

```
representation              id  platform id platspec id language id
--------------------------- --- ----------- ----------- -----------
nameid 1  <string>;         1   3           1           0x0409
nameid 1 3 <string>;        1   3           1           0x0409
nameid 1 3 S L <string>;    1   3           S           L
nameid 1 1 <string>;        1   1           0           0
nameid 1 1 S L <string>;    1   1           S           L
```

A string is composed of 1-byte ASCII characters enclosed by ASCII double quote characters ("). Newlines embedded within the string are removed from the character sequence to be stored.

Strings are converted to Unicode for the Windows platform by adding a high byte of 0. 2-byte Unicode values for the Windows platform may be specified using a special character sequence of a backslash character (\) followed by exactly four hexadecimal numbers (of either case) which may not all be zero, e.g. \4e2d. The ASCII backslash character must be represented as the sequence \005c or \005C and the ASCII double quote character must be represented as the sequence \0022.

There is no corresponding conversion to Unicode for the Macintosh platform but character codes in the range 128-255 may be specified using a special character sequence of a backslash character (\) followed by exactly two hexadecimal numbers (of either case) which may not both be zero, e.g. \83 . The ASCII blackslash character must be represented as the sequence \5c or \5C and the ASCII double quote character must be represented as the sequence \22.

Example (add designer's name that includes non-ASCII characters for Mac and Windows platforms):

```
table name {
  nameid 9 "Joachim M\00fcller-Lanc\00e9";    # Windows (Unicode)
  nameid 9 1 "Joachim Mu\9fller-Lanc\8e";     # Macintosh (Mac Roman)
} name;
```

9.f. OS/2 table

```
table OS/2 {
  FSType <number>;
  Panose <panose number>;
  UnicodeRange <Unicode range list>;
  CodePageRange <code page list>;
  TypoAscender <metric>;
  TypoDescender <metric>;
  TypoLineGap <metric>;
  winAscent <metric>;
  winDescent <metric>;
  XHeight <metric>;
  CapHeight <metric>;
  WeightClass <number>;
  WidthClass <number>;
  Vendor <string>;
} OS/2;
```

Vendor should 4 character-long. If a shorter vendor id is given, it is automatically padded with spaces. A longer vendor id causes an error.

<panose number> is ten (decimal) numbers separated by white space. For <Unicode range list> is a whitespace-separated list of Unicode bit numbers from the OpenType specification for the ulUnicodeRange1-4 in the OS/2 table. <code page list> is a whitespace-separated list of Windows code page numbers from the OpenType specification for the ulCodePageRange1-2 in the OS/2 table. Example:

```
table OS/2 {
  FSType 4;
  Panose 2 15 0 0 2 2 8 2 9 4;
  TypoAscender 800;
  TypoDescender -200; # Note that TypoDescender is negative for descent below the baseline.
  winAscent 832;
  winDescent 321; # Note that winDescent is positive for descent below the baseline.
  UnicodeRange 0   # Basic Latin
   1   # Latin-1 Supplement
   9   # Cyrillic
   55  # CJK Compatibility
   59  # CJK Unified Ideographs
   60  # Private Use Area
   ;
  CodePageRange  1252    # Latin 1
  1251    # Cyrillic
  932     # JIS/Japan
  ;
  XHeight 400;
  CapHeight 600;
  WeightClass 800;
  WidthClass 3;
  Vendor "ADBE";
} OS/2;
```

Note that that for the code page ranges, the list numbers may be separated by any amount of white space. Note that the terminal semi-colon cannot follow a comment character on a line, as all text on a line following the comment character is removed before processing.

9.g. vhea table

```
table vhea {
  VertTypoAscender <number>;
  VertTypoDescender <number>;
  VertTypoLineGap <number>;
} vhea;
```

For example:

```
table vhea {
  VertTypoAscender 500;
  VertTypoDescender -500;
  VertTypoLineGap 1000;
} vhea;
```

9.h. vmtx table

In OpenType, each glyph may have a unique vertical origin y coordinate and a unique vertical advance width. By default, for each glyph the vertical origin y coordinate is set to the value of the OS/2.TypoAscender field, and the vertical advance width is set to the distance between the values of the of OS/2.TypoAscender and the OS/2.TypoDescender. However, other values may be assigned to a glyph as follows:

```
table vmtx {
  VertOriginY <glyph> <number1>;
  VertAdvanceY <glyph> <number2>;
} vmtx;
```

This would result in the glyph's vertical origin Y coordinate and the glyph's vertical advance width being set as shown. The value set here for the vertical origin Y coordinate will also set the topSideBearing value in the 'vmtx' table and the vertical origin y value in the 'VORG' table for the named glyph.

For example:

```
table vmtx {
  VertOriginY \711  864;
  VertOriginY \712  867;
  VertOriginY \713  866;
} vmtx;
```

A special case for the vertical advance width is the set of glyphs referenced by the 'vrt2' feature. The default vertical advance for these glyphs is the horizontal advance of their corresponding target (upright) glyphs. These values will also be overridden by VertAdvanceY values.

10. Specifying anonymous data blocks

The feature file can contain "anonymous" tagged blocks of data that must be passed back to the client of the implementation software. These blocks of data typically contain information needed to specify custom or unsupported tables. The parser will not attempt to parse the data. Each such block is specified as follows:

```
anonymous <tag> {
  # ...
} <tag>;
```

Note: the keyword "anonymous" can be abbreviated as "anon". For example:

```
anon sbit {
  /* sbit table specifications */
  72  % dpi
  sizes {
    10, 12, 14 source {
      all "Generic/JGeneric"
    }
  }
} sbit;
```

The closing brace, tag, and semicolon must all be on the same line to indicate the end of the anonymous block to the parser. White space may be used between tokens on this line, and a comment may follow the semicolon. The "include" directive will not be recognized within the block, starting from "anonymous" and ending at the end of the closing line, so the entire block must exist within the same file.

The data that is passed back to the client starts at the beginning of the line after the opening brace and ends at (and includes) the newline before the closing brace. In the example above, the client is passed back the following data:

```
/* sbit table specifications */
72  % dpi
sizes {
```

```
    10, 12, 14 source {
      all "Generic/JGeneric"
    }
  }
```

along with the tag 'sbit'.

11. Document revisions

v1.11 [4 Sept 2012]:

- Added clarification of backtrack sequence order in chaining contextual lookup rules.

v1.10 [31 March 2010]:

- Fixed typo in example in section 4.d: lookFlag values are separated by spaces, not commas.
- Fixed typo in example in section 8.c on stylistic names; examples: quotes around name string need to be matching double quotes.
- Fixed typo in example in section 5.f.i on Chain Sub rules; second line was missing the target glyph "s".

v1.9 [4 May 2009]:

- Fixed minor typo in Anchor statement, clarified that both mark class and regular glyph class names can be used with the lookupflag keyword.

v1.8 [16 Dec 2008]:

- Revised syntax for mark attachment lookups and the GDEF table
- Revised syntax for the lookupflag statement.
- Added new syntax for contextual rules
- Revised the description of how language systems work.
- Revised syntax to allow stand-alone lookups.
- Added syntax for named value records and anchors
- Added syntax for descriptive names for stylistic substitution features

v1.7 [25 Oct 2006]:

- Fixed incorrect sign for winAscent keyword
- Extended discussion of DFLT script tag and useExtension keyword.
- Fixed minor typos.

v1.6 [28 March 2006]:

- Added discussion of contextual chaining positioning, and how it is supported.
- Extended discussion of size feature
- Fixed minor typos.

v1.5 [23 November 2005]:

- Expanded discussion of languagesystem, script, and language keywords to describe newly added support for DFLT script, and correctly describe handling of 'dflt' lookups.
- Added OS/2 keywords winAscent, winDescent, and hhea keywords Ascender,Descender, and LineGap
- Added supported size table values as decimal fractions as well as decipoints, e.g "10.0" as well as "100" for a point size of 10.
- Added supported OS/2 keyword "fsType" as well as "FSType".
- in GPOS positioning statements, allow single value for value record in a all positioning statements; the case of all but pair positioning statements, this stands for a change to the x-advance of the preceding glyph name.

v1.4 [23 January 2003; supported by HOT library v01.00.36]:

- Added new vmtx table overrides, in order to permit setting vertical metrics for pre-rotated proportional glyphs that are specifically designed and are not simply rotated forms of proportional glyphs.
- Added new OS/2 overrides to set the Unicode and Windows code page range fields:
  - UnicodeRange
  - CodePageRange
- Updated language keywords to be consistent with OpenType spec, i.e using "dflt" instead of "DFLT". Expanded section explaining use of language and script default keywords. Old keywords still work, but cause a warning to be emitted.

v1.3 [23 May 2002]:

- Added support for Vendor tag in OS/2 table

v1.2 [7 March 2001; HOT library v01.00.29]:

- Replaced "except" clause by separate "ignore substitute" or "ignore position" statement; e.g. from "except a @LET sub a by a.end;" to "do not sub a @LET; sub a' by a.end;". (Note that the second rule is now required to be marked to identify it as a Chain Sub and not a Single Sub rule.) "except" clauses will still be handled correctly, but a message will be emitted encouraging users to update the syntax.

(This change was made since the "except" syntax was misleading in that it implied that the exception sequences were exceptions only to the rule at that same statement, whereas in fact they are exceptions to all subsequent rules until the end of the lookup.)

- Value record, anchor, device, contour point, and caret values: changed to a consistent syntax that removes ambiguities and is more human-readable. (They are now enclosed within angle brackets; the "anchor", "contourpoint", and "caret" keywords are introduced; commas now separate device ppem/number pairs from each other.)

Note: Of the above changes, only value record format B in a Single Pos statement is actually implemented. For example, "pos a 80 0 -160 0;" was changed to "pos a <80 0 -160 0>;". Previous syntax will still be handled correctly, but a message will be emitted encouraging users to update the syntax.

- Value records and anchors in GPOS rules (except for the AFM-style Pair Pos format) now immediately follow the glyphs they correspond to; no commas are needed. GDEF LigatureCaret statement: removed commas between caret values. (Implementation not affected, since these sections were implemented.)
- Added ability to handle Single Sub rules with a single replacement glyph for 2 or more glyphs in the target glyph class; e.g. "sub [one.fitted one.oldstyle one.tab.oldstyle] by one".
- Added support for named lookupflag attributes e.g. "lookupflag IgnoreLigatures;".
- Updated the OT layout algorithm pseudocode.
- Clarified that a pair positioning rule is treated as a class pair if and only if at least one glyph class is present, even if that class is a singleton glyph class.

v1.1 [1 December 2000; HOT library v01.00.28]:

- Added support for "languagesystem" statement. Note that this entailed removing support for script, language, and named lookup blocks/statements in the 'size' feature, and removing support for script and language statements in the 'aalt' feature.
- Added implementation note at end of Syntax: Glyph Classes section.
- Updated OTL engine algorithm and lookup ordering section.
- Specified that the 'vmtx' table overrides are recorded in the 'VORG' table as well.

v1.0 [29 September 2000; HOT library v01.00.24]:

- Added useExtension keyword support for lookup blocks and the 'aalt' feature block.

v0.9 [25 April 2000; HOT library v01.00.23]:

- Added vmtx table keyword VertOriginY.
- Changed description of 'aalt' feature to describe sorting of alternate glyphs.
- Added description of head.fontRevision setting and reporting.
- Changed syntax for feature parameters.

v0.8 [24 February 2000; HOT library v01.00.23]:

- Added syntax for feature parameters.
- Various miscellaneous description changes.

v0.7 [11 October 1999; HOT library v01.00.22]:

- Added grammar for completely specifying GSUB, GPOS, BASE and GDEF: full-fledged value records, device tables, anchors, required features, GPOS LookupTypes 3-6 and 8, generalized GSUB LookupType 6, subtable breaks. Indicated all parts not currently supported in [brackets].
- 'aalt' creation: specified that 'aalt' will be created after all other features; special script rules.
- Added OS/2.FSType, WeightClass and WidthClass overrides.

v0.6 [22 March 1999; HOT library v119]:

- Added subtable keyword.

v0.5 [29 January 1999]:

- Added name table overrides.

v0.4 [20 January 1999]:

- HTML-formatted the document

- Updated glyph name attributes
- MM metrics can be represented by a single number if constant across masters
- OS/2.XHeight,CapHeight,TypoLineGap overrides
- hhea.CaretOffset override
- vhea.VertTypoAscender,VertTypoDescender,VertTypoLineGap overrides

v0.3 [9 October 1998]:

- Revised 'aalt' creation algorithm.
- Revised table field override support: OS/2.Panose, OS/2.TypoAscender, OS/2.TypoDescender, head.FontRevision.
- Added chaining contextual substitutions.
- Added support for anonymous data blocks.
- Changed glyph class expansion rules.

v0.2 [18 March 1998]:

- Changed keyword "replace" to "substitute" (or "sub"); introduced keyword "position" (or "pos").
- Added section on ordering of lookups and rules
- Added lookupflag attribute.
- Expanded Syntax section: lists of keywords, special characters, and glyph name, glyph class name and lookup label name restrictions.

v0.1 [6 February 1998]: First version