Graphite Description Language
Version 3.000

M. Hosken, B. Hallissy, W. Cleveland, S. Correll, A. Ward
SIL Non-Roman Script Initiative (NRSI)

1 Introduction

This document is intended to be a semi-formal description of the Graphite description file format known as Graphite Description Language (GDL). As such, it introduces concepts in, hopefully, a more natural way than a formal description would require.

This document is also primarily concerned with ensuring that the language has adequate descriptive power to describe all possible script behaviors. Therefore the document will tend to concentrate on the more complex aspects of script description, thus not representing the balance that you would find in a typical description in which most behaviors are relatively straightforward.

This document has been written assuming a basic understanding of fonts, characters, glyphs, and rendering issues. Please refer to the Glossary in section 9 for a definition of terms.

Please send comments or questions to: graphite_nrsi@sil.org.

1.1 Capabilities of the Graphite system

The Graphite system is designed to handle the following kinds of complex rendering situations:

- shifting and kerning, where the position of a glyph is adjusted based on the presence of a neighboring glyph
- ligature substitution, where one glyph is used to represent several underlying characters
- reordering typical of Indic scripts, where the order of the rendered glyphs is different from the order of the corresponding underlying character data
- stacking diacritics, using attachment points
- bidirectionality, as found in Hebrew and Arabic-based writing systems

See the “Graphite Requirements” document for more details.

1.2 Graphite and Unicode

Graphite is intended to be used with Unicode data, that is, in situations where the underlying data complies with the Unicode standard. Similarly, a font used to render with Graphite should be Unicode-based; that is, the character values in the font’s cmap should be Unicode codepoints. Although it is possible to use Graphite with “hacked” or customized encodings, this is not recommended.

2 Overview: rules, glyphs, and passes

2.1 Rules

The basis of a rendering description is rules. Rules are used for almost everything, and allow for contextual attribute assignment, substitution, etc. The rules are replacement-type rules in a format familiar to those with experience in generative phonology.

Our initial discussion will concentrate on substitution type rules. There are other types which we will come to.

A typical substitution rule might be:

```
gLowercaseI > gDotlessI / _ gTilde;
```

This rule says that an underlying lowercase I glyph is replaced by a corresponding dotless I glyph on the surface, when followed by a tilde glyph. More precisely a glyph stream containing:

```
... gLowercaseI gTilde ...
```

would be modified to contain:

```
... gDotlessI gTilde ...
```

The names used for the glyphs are identifiers that are assumed to have been defined to refer to particular glyphs in a font. The particular mechanics of how this is done will be covered in a later section.

From the above example, we see that a rule consists of three parts: the *left-hand side*, the *right-hand side*, and the *context*. The left-hand side (or *lhs*) consists of the particular glyphs in the underlying form that are to be replaced. The right-hand side (*rhs*) gives the glyphs that will replace the glyphs in the lhs. Notice that there is a strict one to one correspondence between the glyphs on the lhs and on the rhs. Following the `/` is the *context* which describes the environment in which the *lhs* is to be located and the *rhs* will be output.

The various parts of the rules—lhs, rhs and context—should not be seen as strings, but as sequences of glyphs. Thus, in the above rule we are saying that the glyph `gLowercaseI` is being replaced by the glyph `gDotlessI` when followed by the glyph `gtilde`. In the context, therefore, `_` is used to represent a single glyph corresponding to a glyph in the lhs of the rule.

Specifying both the rhs and context provides the greatest clarity and self-description when describing rules in terms of matching; however, it is possible to use a simpler format for rules which has no context. For example, the above rule could have been written:

```
gLowercaseI gTilde > gDotlessI gTilde;
```

This is not as clear since it does not highlight the glyphs being changed. It is also weaker in not allowing the `gTilde` to be re-matched in the same pass. But the rule is possible and an optimizing compiler (which we are not promising to develop) should give the same results. This rule is strictly equivalent not to the first rule but to

```
gLowercaseI gTilde > gDotlessI gTilde / _ _ ;
```

Notice the two _, one for each glyph on the left hand side of the rule. It is an error if there are a different number of _ in the context as there are glyphs on the left- and right-hand sides.

The context of a rule can be as complex as needed. The lhs does not have to refer to a contiguous sequence of glyphs:

```
gLowercaseI gTilde > gDotlessI gTilde / _ gLowerDia _ ;
```

Notice that the two _ in the context correspond to the two glyphs in the rhs, and also to the two glyphs in the lhs.

Some rules do not perform substitutions at all, but only set *attributes* on items in the glyph stream. In such cases, the left-hand side (and the right angle bracket) are omitted:

```
gCapA {kern.x = -KernPostV} / clsCapVW _ ;
```

The code inside the curly braces sets attributes on the capital A glyph, kerning it towards the capital V or W. `KernPostV` is defined elsewhere in the file as a glyph attribute, a numeric constant for this glyph.

Rules without a lhs are used particularly in the positioning table, which will be discussed in more detail further on.

Semi-colons are required to terminate rules. Line continuation is assumed if there is no semi-colon. Other statements may use an optional semi-colon terminator. (For the technically minded, semi-colons are actually separators.)

Comments are preceded by two slashes.

```
a > b / _ c;  //this is a comment
```

Note that comments do not require semi colons.

2.1.1 Classes

If every individual combination of glyphs that we want to alter had to be spelled out with its own rule, then the description would be impossibly long. Instead a system of glyph classes is available. Our first rule can be generalized to the following:

```
clsDotted > clsDotless / _ clsUpperDia;
```

This rule says that all dotted glyphs (`i`, `j`, etc.) are replaced by their dotless counterparts when followed by an upper diacritic.

From this we see that classes are a bit like arrays. When an element from a class is matched, its position in the class is remembered so that it can be used to refer to an element from a different class (which must be the same size or bigger). This correspondence is very helpful to reduce the number of rules.

The use of `cls` to prefix the class name is purely a coding convention. It is used in this document to aid in rule readability.

A more complex substitution rule might change two glyphs at once:

```
clsCons clsVowel > clsConsJoin clsVowelDia / _ ZWJ _ ;
```

This rule might occur in an Indic script where vowels may be diacritically joined to the preceding consonant via a zero-width joining character (or glyph in this context). This rule is not ideal since it would probably be preferable to delete the ZWJ at the same time. We will come to that later.

2.1.1.1 Variables & Lists

Classes are defined using a standard assignment command (in the glyph table). Assignments allow for variables to be defined either as individual values or as lists. Assignments are of the form:

```
variable = value;
```

In the case of a list, the list is identified between `()`. Thus:

```
clsDottedI = (gLI, gLBarredI);
clsIWidth  = (clsDottedI, gLL, gUI, gUBarredI);
```

Commas within a list are optional. Also note that the semi-colon following a class definition or variable assignment is optional, unlike in rules where it is required. Class names do not need to be declared separately, unlike many programming languages; the assignment statement functions as the declaration.

Elements can be added to the end of the list using the += operator. For example:

```
clsDottedI += gLJ;
clsIWidth += (gUJ, gUL);
```

The list mechanism also allows for temporary unnamed "classes" within rules, although this is not good practice since it does not encourage the definition to be self-documenting. By naming every glyph class, the GDL author is giving documentation to their description as they go. There are other mechanisms to encourage this throughout the file.

An example of a temporary list (if it must be used) is:

```
clsDotted > (gLDotlessI, gLDotlessBarredI, gLDotlessJ)
   / _ clsUpperDia;
```

2.1.1.2 Ranges

Lists may also be made up from ranges. A range is an inclusive list, including both endpoints. Thus:

```
clsCaps = unicode(0x0041 .. 0x005A);
```

is equivalent to:

```
clsCaps = (unicode(0x0041), unicode(0x0042), unicode(0x0043),
      unicode(0x0044), …, unicode(0x005A));
```

Both forms would create a class containing the glyphs for all the uppercase letters in the standard Roman alphabet. The `unicode` and related functions will be discussed in more detail later.

2.1.1.3 The ANY class

A special class, called "ANY," can be used to match any glyph. This class also has a special use within the Graphite system.

2.1.2 Attributes

In addition to substituting one glyph for another, rules may be used to associate information with the glyphs that have been matched by the rule. This information is stored in attributes. For example:

```
gCapA {kern.x = -bb.width/10} / clsCapVW _ ;
```

indicates that a capital A following V or W should be kerned inwards by 10% of the bounding box of the A. Notice that for this rule, there is no lhs. Since the lhs and rhs are the same, we do not need the lhs. This is because there is no substitution occurring. In fact we could do away with the context also with:

```
clsCapVW  gCapA {kern.x = -bb.width/10};
```

`The bb.width` expression is one of several read-only glyph metrics that can be referenced to aid in positioning.

Multiple attributes may be assigned within one rule, as in:

```
clsBase clsMark {shift.x = -10m; shift.y = ascent / 2};
```

This example is shifting a "mark" glyph 10 units to the left and up almost half a line. Attributes with subfields can used a structured syntax; for instance, the above rule can also be written:

```
clsBase clsMark {shift {x = -10m; y = ascent / 2}};
```

There are a number of different attributes which a particular element may have, most of which affect positioning. They are all considered in later sections.

2.1.2.1 Types of attributes

There are two kinds of attributes, *glyph attributes* and *slot attributes*. The values of glyph attributes are constant with respect to a glyph's ID number; they do not depend on the glyph's position in the data or any neighboring glyphs. For instance, each letter 'A' in the text would have exactly the same glyph attribute values. Glyph attributes can be thought of as extensions to the glyph metrics in the font. They are set in the glyph table, and will be discussed more fully below.

Only slot attributes are set in rules, and therefore may have different values depending on which rules have been fired during the process of rendering a specific glyph. In our examples above, the `kern.x`, `shift.x`, and `shift.y` attributes are slot attributes. Not every capital V, W, and A would have the same values for these attributes; only the ones that occur next to each other and so cause the example rule above to fire would have adjusted values for the `kern.x` attribute. Similarly the "base" and "mark" glyphs would have different values of `shift.x` and `shift.y` depending on whether they were involved in the firing of the second example rule.

2.1.3 Optionality

One of the most useful things about regular expressions is the ability to have optional elements, which are elements which may or may not occur. They are marked by a `?`. GDL uses the same character `?` to mark optional items. Thus the rule for dotless i may be extended to match with an optional lower diacritic coming between the dotted i and the upper diacritic:

```
clsDottedI > clsDotlessI / _ clsLowerDia? clsUpperDia;
```

Optional items may occur in the context, as shown above, or on the left-hand side of a rule, but not on the right-hand side:

```
clsVowel clsTone? > clsUpperVowel clsUpperTone / clsCons _ _ ;
```

In this case, if `clsTone` finds no match, then no output from the corresponding element on the right-hand side, `clsUpperTone`, is generated.

Graphite also provides the capability to mark element sequences as optional. To group the elements, use `[ ]`. For example:

```
clsDottedI > clsDotlessI / _ [ clsLDia gLower ]? clsUpperDia;
```

uses the dotless i even if the sequence of grouped elements appear between the dotted i and the upper diacritic.

Notice that the other regular expression string operators: `*` and `+` are not supported for two reasons: they would confuse context referencing, and they are unbounded. But it is possible to provide a limited form of these operators using `[]`:

```
[x [x [x [x]? ]? ]? ]?
```

is equivalent to `x{0,4}` in Unix regular expression syntax.

For more information on how optional rules work, see the sections under Data Processing.

2.1.4 Rule Constraints

In addition to setting attributes on a matched element, rules can be conditionally executed based on the attributes of an element.

```
gB {kern.x = MAXSP - @1.rsb - @2.lsb} / gA _ {@1.rsb + @2.lsb > MAXSP};
```

This rule will only be applied if its context matches the glyph stream *and* the constraint in the rule's context is satisfied. For this rule, if the physical gap between the first and second rule elements is greater

than a given value, the rule will be executed, which will set the interglyph gap to equal a fixed value. Constraints within a rule can only be specified in the context and the context must be explicitly stated.

Constraints for one or more rules can also be specified using *features*, which we will come to later.

2.2 Glyph Identification

So far we have described glyphs by name. But how are these names converted to actual glyph numbers in the font?

There are four ways of getting hold of a glyph number:

- by using the actual internal glyph number in the font;
- by Unicode value via the internal character map (cmap) in the font, which takes a Unicode codepoint number and returns a glyph number;
- by Postscript name; and
- by 8-bit character code according to a codepage and then via the font character map.

Each of these methods has its own strengths and weaknesses.

2.2.1 Glyph ID

Glyph IDs are identified numerically using the following syntax. Notice that the number may be in decimal or hexadecimal.

```
glyphid (439)     glyphid(0x1B7)
```

The GlyphID command can also take a list of values which it returns as a list. Thus:

```
glyphid(0x1b7, 23, 128)
```

The advantage of the `glyphid` command is that you have direct access to any glyph in the font even if it has no usable Postscript name and it does not appear in the `cmap`. The difficulty is that glyph IDs are often unique to a particular font and even a particular version of the font. Using glyph IDs directly requires close liaison with the font designer.

2.2.2 Unicode

A glyph is identified via its Unicode value using the following syntax:

```
unicode (0x203F)     unicode(8255)        U+203F
```

Care should be taken with hexadecimal numbers, which are often used for Unicode codepoints, but which must be explicitly marked in this syntax. Thus the second value here is not 0x8255 but 0x203F. However, a 0x is not needed when using the U+ syntax. Thus all of the examples shown above are equivalent.

The `unicode` command may also be used to generate a list just as in the `glyphid` command.

Using the `unicode` command to identify a glyph can be very powerful. It has the advantage over `glyphid` of not being dependent upon font and version. But it does require that the glyph be identified in the cmap of the font, and therefore is only useable on such “exposed” glyphs.

2.2.3 Postscript

Accessing glyphs via their Postscript names is an important and powerful method. Since it is possible to give every glyph in the font a unique name, this method allows the description file to refer to glyphs that do not necessarily have Unicode values in a cmap.

There is also a weakness with this approach. It is not necessary, when designing a font, to give every glyph a unique name, and so this method may not always be able to identify every glyph in a font.

The primary context for this method is where a script engineer is working with a font designer and they can agree on names for glyphs. This frees them to work semi-independently, without the need for the font designer to take great care over glyph numbering. So long as the names line up, everything should be OK.

A glyph is referred to by its Postscript name using this syntax:

```
postscript ("Ccedilla")
```

2.2.4 Codepoint

The final method is very similar to the Unicode method. This is to give an 8-bit codepoint value which is mapped through a codepage mapping from its 8-bit value to a 16-bit Unicode value and thence to the glyph ID via the cmap. Due to its similarity to the `unicode` command, it inherits all its strengths and weaknesses.

Eight-bit values can be entered using two methods. The parameter to `codepoint()` can be a string, in which the characters are converted to 8-bit values and thence to Unicode then glyph ID. Alternatively a number can appear (decimal or hex).

```
codepoint ("a")      codepoint(192)       codepoint(0xC0)
```

The codepage to use in conversion is specified by defining the `CodePage` directive. Any 8-bit conversions from then on will use that codepage. The default codepage used at the start of the file is codepage 1252.

It is also possible to specify a particular codepage within the `codepoint()` command as the second parameter:

```
codepoint("a", 1251)
```

The `codepoint` command may also be used to generate a list if it is given a list or a string as its first parameter:

```
clsDia = codepoint((0x93, 0x94, 0x95));
clsVowels = codepoint("aeiouAEIOU");
```

The standard C character escape codes are allowed: `\t` (tab), `\n` (newline), `\\` (backslash), etc.

2.2.5 Glyph class identifiers

The examples of rules we have seen so far have not used any of these approaches to glyph referencing, and yet they are legal. Why is this so?

Rather than having to use a full glyph identification for every reference to a glyph, or codepoint, it is sensible to use identifiers to save effort and to improve readability.

Thus some of the above examples might have been entered for IPA93 as:

```
gI       =  codepoint("i");
gBarredI    =  codepoint(0xF6);
gJ       =  unicode(0x006A);
gL       =  codepoint("l");
gUI         =  unicode(0x0049);
gUbarredI      =  postscript("UCBarredI");
gUJ         =  unicode(0x004A);
gUL         =  unicode(0x004C);
gDotlessI      =  codepoint(34);
gDotlessBarredI   =  codepoint(0xAA);
gDotlessJ      =  codepoint(0xBB);

clsLower    = (gI, gBarredI, gJ, gL, gDotlessI,
         gDotlessBarredI, gDotlessJ);
clsUpper    = (gUI, GUbarredI, gUJ, gUL);
clsBarred   = (gBarredI, gUbarredI, gDotlessBarredI);
clsDotless  = (gDotlessI, gDotlessBarredI, gDotlessJ);
```

It is unlikely that anyone would use such a wide variety of referencing schemes in the same file, but notice how much clearer it is to refer to glyphs within a description file using identifier names.

These assignments would be done in the glyph table. Notice that the list parentheses are not needed for single glyphs.

2.2.6 References

It can be awkward to constantly have to keep naming everything. An alternative is to use the `@` to refer to the corresponding glyph on left hand side. Thus the following rule simply copies a glyph:

```
clsCons > @;
```

A more common use is to reference glyphs in the context by number:

```
clsA clsB > @2 @1;
```

This rule swaps the glyphs.

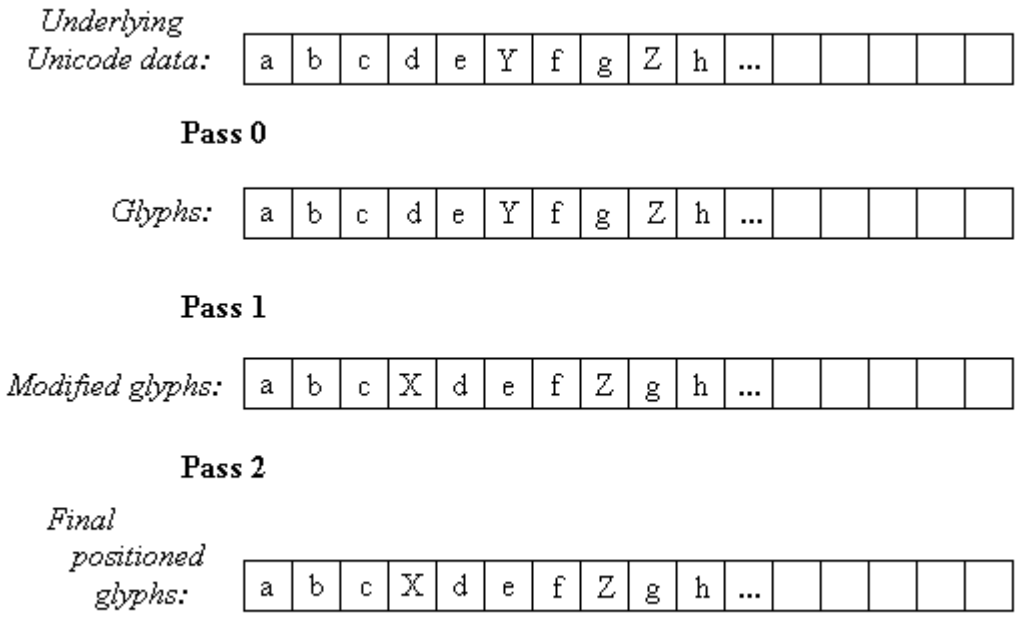
Notice that `@` may only be used on the right hand side of a rule. It cannot refer to an optional element. We will see much more of `@` later on.

2.3 Passes, streams, and slots

Graphite processing is organized into a sequence of two or more passes. Each pass takes a stream of glyphs as input, processes its contents, and produces an output stream. This output stream then serves as the input to the following pass. The initial pass (considered pass zero) converts Unicode characters into glyphs. The other passes run rules, performing matching on the input stream and placing the results of their rules into the output stream. In particular the final pass places the glyphs into their final positions for rendering.

Streams are made up of a sequence of slots, each containing a single glyph. There is a correspondence between the slots in the input stream and those in the output stream, and slots can be inserted, deleted or rearranged.

The figure below shows an example of passes and the slot streams they process. Each square represents one slot, and holds one glyph. Notice that during pass 1 an ‘X’ was inserted (between the ‘c’ and the ‘d’), so it is appropriate to think of a slot being inserted into the stream to hold it. Similarly, ‘Y’ has been deleted, and the ‘Z’ has been reordered—moved with respect to a neighboring slot (the ‘g’).



Pass zero is automatically handled by the Graphite engine. The other passes, which contain rules, are defined within the GDL program.

Passes are organized into tables. There are four kinds of tables that can include rules: `linebreak`, `substitution`, `justification`, and `positioning`; of these the substitution and positioning tables are the most commonly used. The following shows an example of how the tables and passes of a GDL program might be organized:

```
table(substitution)

pass(1)
// rules to handle ligatures
endpass;

pass(2)
// rules to merge base characters and diacritics
endpass;

endtable;   // end of substitution table

table(positioning)
// rules to attach glyphs
endtable;
```

Tables and passes are discussed in more detail in the following sections.

3 File Structure

The GDL file is made up of a set of tables. Currently there are eight table types, one each for:

- feature definitions
- language definitions,
- glyph definitions
- line-breaking rules
- substitution rules
- justification-related substitution rules
- positioning rules, and
- name definitions.

The linebreak, substitution and positioning tables are used in particular phases of the rendering process. Within them, the rules can be grouped into multiple, ordered passes. Rules can also be conditionally applied based on features which are defined in the feature table and which are referenced by conditional statements in the rules.

The glyph table is used to define the glyph classes and provide information about the glyphs that the rules will later use. The line-break table can be used to provide information on how lines should be broken. The name table provides a way for arbitrary text strings to be stored in a compiled GDL file. There are also several global settings and directives that can be applied across various sections of a file.

3.1 C Pre-Processor

To allow for commenting and some sophisticated macros and definitions, the description file is first passed through the C pre-processor. For example, the C pre-processor allows a standard file to be included which gives definitions for glyphs:

```
#include <IndicGlyphs.gdh>
```

There are many other uses, including using the same file for different encodings. Organizing everything in tables facilitates including files. For example, even if the #include statement occurs in the midst of the substitution table, the various table types in the included file will be properly interpreted.

Note: any paths inside an `#include` statement must use Linux-style forward slashes rather than Windows-style backslashes. Paths should be relative to the current working directory (not the main GDL file or the including file).

Graphite has a complex description language so that it can describe all the different vagaries of the orthographies of the world. The needs of one group of orthographies can be very different from the needs of another group. Using the C pre-processor's macro capability allows us to develop macro sets which will make particular common features of an orthography family easier to describe. The added burden of learning particular sets of macros for particular needs will be offset by their ease of use. But different regions will probably have different macro sets. For example, some possible first candidates for macros would be ligature representation and more complex Arabic rules.

The very beginning of the file may likely contain various pre-processor commands, such as:

```
#define cpt    codepoint
#define u (x)  unicode (x)
#define ps     postscript
#define gid    glyphid
#define str    string
```

```
#define LG_USENG 0x4090    // US English
#define CP_USSTD 1252      // US standard code page
```

3.1.1 Standard Include File

There is a standard `#include` file provided with Graphite, which provides easy access to numerous identifiers. This file provides #defines for the GDL constants and abbreviations seen through out this document, such as `sub` for `substitution` and `DIR_LEFT` to indicate left-to-right. To use this standard #include file, place the following statement at the beginning of your GDL program:

```
#include “stddef.gdh”
```

3.2 Global Settings

There are a number of global settings that are typically used at the beginning of a file.

3.2.1 AutoPseudo

```
AutoPseudo = 1; // default
```

This controls *auto-pseudo* glyph mapping, which is an advanced feature used when dealing with multiple Unicode codepoints mapping to the same glyph. See the discussion in the Advanced Concepts section. This has global scope and is used on a line by itself at the beginning of the file. If it is set multiple times (typically with `#include` files) all values must agree.

3.2.2 ScriptDirection

```
ScriptDirection = HORIZONTAL_LEFT_TO_RIGHT;  // default
```

This variable indicates the directionality of the writing system. Possible values are:

```
HORIZONTAL_LEFT_TO_RIGHT
HORIZONTAL_RIGHT_TO_LEFT
VERTICAL_FROM_LEFT
VERTICAL_FROM_RIGHT
```

(Vertical scripts are currently not supported by the engine.)

It is possible that some GDL implementations may be appropriate for more than one direction, in which case the values can be added together:

```
ScriptDirection = HORIZONTAL_LEFT_TO_RIGHT + HORIZONTAL_RIGHT_TO_LEFT;
```

In this situation it is the responsibility of the calling application to determine the writing system direction.

3.2.3 ScriptTag

```
ScriptTag = (“ABC1”, “ABC2”);
ScriptTag += “ABC3”;
```

This variable stores information about the script being implemented by the file. Since a given file can describe more than one script, the setting accepts a list of values. This list can be appended to using the `+=` operator. Script tags must be strings not longer than four characters.

3.2.4 Bidi

```
Bidi = true; // default
```

This setting is used to indicate whether or not a pass to run the Unicode bidirectional algorithm should be included. The value is true by default, but may be set to false as an optimization for scripts that have

no internal bidirectionality (which is true of most scripts except Arabic and Hebrew). The concept of internal bidirectionality and the bidi algorithm are discussed in a later section.

3.2.5 ExtraAscent

```
ExtraAscent = 200m;
```

Due to the fact Graphite can position glyphs in complicated ways, it is possible that the ascent as defined in the original font is not appropriate for the Graphite renderer. For instance, if Graphite provides for a stacking diacritic mechanism, it may be helpful to increase the ascent of the font to allow vertical space likely to be needed for the diacritics. The `ExtraAscent` global can be used for this purpose.

Including an 'm' after the value of the global means that the number is scaled relative to the size of the em square as defined by the MUnits directive. This feature is discussed more fully in the section on metrics.

The default value for `ExtraAscent` is zero.

3.2.6 ExtraDescent

```
ExtraDescent = 100m;
```

The `ExtraDescent` global can be used similarly to `ExtraAscent` to provide for the fact that a Graphite renderer may adjust the vertical position of glyphs so that they extend below the standard descent as defined within the font.

The default value for `ExtraDescent` is zero.

3.3 Tables

A GDL file uses tables to organize assignments and rules. A table is identified by starting with `table()` and ending with `endtable`. For example:

```
table(substitution)
/* rules */
endtable;
```

introduces the substitution table of rules, which are used to reorder, substitute, insert, and delete glyphs before positioning. Terminating semi-colons are optional for both commands. There are seven table types indicated by `feature`, `glyph`, `name`, `linebreak` (or `lb`), `substitution` (or `sub` or `subs`), `justification` (or `just`), and `positioning` (or `pos` or `position`).

It is not necessary to group all the elements in a table together in the file. For example, you may interleave two tables so that semantically similar rules from different tables can be near each other in the file. This will result in multiple `table` commands referring to the same table. The compiler will collect all these separated elements together and sort them out. Features, classes, glyph attributes, etc. must be defined before they are used in rules; the tables where these are defined are described in later sections.

An `endtable` command is required to indicate the end of a table. If a new `table` command is encountered before an expected `endtable`, the statements for the new table are processed and the next `endtable` statement causes a return to the previous table:

```
table(substitution)
/* rules for the substitution table */

table (positioning)
/* rules for the positioning table */
endtable;  /* ends the positioning table */
```

```
/* more substitution rules */
endtable;  /* ends the substitution table */
```

Notice that this has the effect of syntactically nesting one table in another, but semantically the two are independent. This nesting capability is helpful when using #include files.

3.3.1 Passes

The tables that contain rules—the linebreak, substitution and positioning tables—are made up of one or more passes. Passes are identified by a `pass()` statement parameterized by a number. Note that the number is relative to the table containing the pass, not to the overall process. The `endpass` statement terminates a pass. The `pass` statement is optional for tables with only one pass.

As with tables, if a new `pass` command is encountered before an expected `endpass`, the statements for the new pass are processed and the next `endpass` statement causes a return to the previous pass:

```
pass(1)
/* rules for pass 1 */
pass(2)
/* rules for pass 2 */
endpass;
/* more rules for pass 1 */
endpass;
```

If no `pass` statements have been encountered for a table type, the pass is 1. The current pass for a given table type is remembered. If the table type changes, the current pass number for this new table type is used if one has been set.

```
table (sub)
pass (2)
/* rules */
table (pos)
pass (3)
/* rules */

table (sub) // from #include file
/* these rules go in pass 2 */
endtable;
table (pos)
/* these rules go in pass 3 */
endtable; // end #include file

/* this is position table pass 3 */
endpass; // pass 3
endtable; // position
/* this is substitution table pass 2 */
endpass; // pass 2
endtable; // substitution
```

All rules must be in a pass. If no `pass` statement is encountered in a table, all rules are placed in pass 1. For a multi-pass table, all rules must be explicitly placed in a pass.

3.4 Environment

Directives allow the author to specify how certain statements are interpreted. The directives are applied across various sections of the file with the `environment` (or `env`) and `endenvironment` (or `endenv`) keywords. Environment statements can span multiple tables. Directives can also be applied at the

beginning of tables or passes. This effectively creates a new environment that ends with the table or pass.

Here is an example showing how a file might be organized using the major structural elements of GDL (table, pass, environment).

```
AutoPseudo = 1;
environment {CodePage = 1252; MUnits = 1024; PointRadius = 6m}

/* feature and name tables may go here */

table (glyph) {AttributeOverride = true};
/* classes defined, glyph attributes set */
endtable

table (linebreak);
/* set breakweight preferences */
endtable

table (sub) {MaxRuleLoop = 3};
pass (1) {MaxRuleLoop = 5}; // this value overrides the table value
/* rules for substituting, reordering, inserting, and deleting */
pass (2); // this uses MaxRuleLoop = 5 since nested in pass 1
/* more rules /
endpass; // pass 2
endpass; // pass 1
pass (3);
/* rules using MaxRuleLoop = 3 for table*/
endpass;
endtable; // substitution

table (pos);
/* positioning rules */
endtable;

/* … */
endenv
```

The `environment` statements can also be used for a subset of rules within a table or pass. When a new environment begins the previous directive values are saved (pushed), and when that environment ends the previous values are restored. The simplest way to specify the directives for an entire file is to place an `environment` statement before all tables and an `endenv` statement at the end of the file. Any included files can provide their own environment which can be popped when the include file ends. A default environment containing the default values for the directives is present if no explicit environments are in scope.

Tables may have directives applied within them.

```
table (glyph) {CodePage = 1252; MUnits = 1024};
/* class definitions */
endtable;
```

This is equivalent to:

```
table (glyph);
env {CodePage = 1252; MUnits = 1024};
/* … */
endenv;
endtable;
```

If tables are nested then the directives are pushed before the inner table is entered and popped after the inner table is ended.

Passes can also have directives associated within them, though typically only `MaxRuleLoop` is relevant. The mechanism is the same as for tables.

3.4.1 Directives

There are a number of directives that affect how certain statements are interpreted. The value shown in the example indicates the default. They can be applied with the `environment`, `table`, or `pass` statements.

3.4.1.1 AttributeOverride

```
AttributeOverride = true;
```

This controls how conflicting glyph attributes are resolved. It is discussed in the section on the glyph table.

3.4.1.2 AutoKern

```
AutoKern = false;
```

The auto-kern algorithm is part of the automatical collision avoidance mechanism.When `AutoKern` is set to true, automatic kerning will be performed to avoid collisions between sequences. The `collision.flags` bitmap attribute should be set to include the value 16 for glyphs that should be kerned. Automatic collision avoidance is discussed in the Advanced Concepts section.

3.4.1.3 CodePage

```
CodePage = 1252;
```

This assignment allows the redefinition of the default codepage used in `codepoint` and `string` commands (discussed later).

3.4.1.4 CollisionFix

```
CollisonFix = 0;
```

When the value of `CollisionFix` is something other than zero, it indicates that automatic collision avoidance adjustments should be performed. The value of the directive specifies how many iterations the algorithm should use. Automatic collision avoidance is discussed in the Advanced Concepts section.

3.4.1.5 MaxBackup

```
MaxBackup = 0;
```

This indicates the amount by which the rules in a pass can cause the stream to back up. The use of this directive is discussed in the Advanced Concepts section. Note that to use the back-up mechanism you will mostly likely need to set MaxRuleLoop to about twice the value of MaxBackup.

3.4.1.6 MaxRuleLoop

```
MaxRuleLoop = 5;
```

This limits the number of rules that can be applied without advancing the slot position in the input stream and is used for avoiding infinite loops. It is discussed more fully in the section on scan position.

3.4.1.7 MUnits

```
MUnits = 1000;
```

This directive specifies how many units are in a font's em square. To scale an integer using this quantity, postfix an ‘m’ to it. Scaled numbers must be used when specifying the coordinates for attachment points

and ligature components and for adjusting glyph positions. This is discussed more fully in the section on metrics.

3.4.1.8 PointRadius

```
PointRadius = 2m;
```

This controls the default value used for finding points actually on a glyph given coordinates for a point close to the glyph. It is discussed more fully in the section on attachment points. It must be a scaled number.

3.5 Glyph Table

The glyph table is where glyph classes and glyph attributes are defined. The glyph table has the following syntax:

```
table (glyph) { /* directives */ }

/* class definitions and glyph attribute assignments */

endtable;
```

Recall from Section 2.2.5 that classes are defined using a standard assignment command. Classes can be defined with just one element, a list of elements, or a range of elements:

```
clsCapitalX = gCapX;

clsDottedI = (gLowercaseI, gLowercaseBarredI);
clsIWidth  = (clsDottedI, gLowercaseL, gUppercaseI, gUpperBarredI);

clsCaps = unicode(0x0041 .. 0x005A);
```

A typical *glyph attribute* is an attachment point, which specifies where to connect two glyphs together (typically a base character and diacritic—each would have an attachment point). Glyph attributes are used to define such points since they do not depend on the glyph's location in the glyph stream. (In a positioning rule, if a base character followed by a diacritic is found, *slot attributes* are then set specifying that these two particular instances of the glyphs are to be joined. These slot attributes apply only to glyphs that occur in a given slot or position in the glyph stream.)

Glyph attributes are frequently defined for an entire class, which effectively sets the attributes for every glyph in the class. Since it is possible for a glyph to be in more than one class, it is also possible for a glyph attribute of a given glyph to be set to different values. The `AttributeOverride` directive is used to determine whether the first or last value is used. If this directive is false, then overriding doesn't happen so the first value will be used. If it is true (default), then the last value is used.

There are two ways of assigning a glyph attribute. First, we can use normal variable assignment as in:

```
clsBase.udap = point(advancewidth/2, bb.top + bb.height/10);
```

Second, we can use the attribute assigning mechanism:

```
clsBase {udap = point(advancewidth/2, bb.top + bb.height/10)};
```

(We'll discuss points more fully in the next section.) Note that the above statement could be mistaken for a rule with no lhs or context, but it is known to be a glyph assignment statement because it occurs in the glyph table. No floating point numbers are allowed in a GDL file.

Like class names, user-defined glyph attribute names do not need to be declared separately.

The system-defined glyph attributes include directionality, line break weight, and ligature component metrics, as well as standard glyph metrics available from the font. In addition the author can create his own glyph attributes as with the `udap` attachment point in the above example and the `upperloc` variable below. These are specified with user-defined names and can contain an integer value.

```
clsUpperDia = (gCaron, gUmlaut, gAcute, gGrave) {upperloc = 850m};
```

This `upperloc` value could later be referenced in positioning rules to provide a consistent height for these diacritic glyphs.

In a rule, glyph attributes can be accessed either for the current slot being operated on or for other slots by prefixing the attribute name with the reference operator (`@`) and a slot number. For example:

```
clsBase clsUpperDia {shift.x = -@1.advancewidth/2; shift.y = upperloc};
```

sets the `shift.x` attribute of all the glyphs in `clsUpperDia` to half the `advancewidth` value of the glyphs in `clsBase`.

3.5.1.1 Attachment Points

As part of the glyph attributes, it is possible to define named points which can then be used to set attachment (slot) attributes. In a TrueType font, *points* which specify a glyph's contours are organized into *paths*. A path can contain just one point. All points and paths are numbered. For technical reasons the point numbers can be difficult to use; however; the path numbers are quite convenient and can easily be obtained by the font designer (using a program like Fontographer®).

There are three ways to describe attachment points in GDL. They all involve specifying an offset from a *base* point. Ideally the base point has actually been designed into the font for attachment purposes. The point in question should be the only one in its path. The `gpath` function is used to specify the first (or only) point in a path.

```
gA { udap = gpath(3, 0, bb.height/10);
     ldap = gpath(4) }; //no offset is required
```

The first argument is the number of the path, the second and third (if present) are the x and y offsets. Of course if the offsets are omitted, they are assumed to be zero. In this example `udap` (and `ldap`) are short for upper (or lower) diacritic attachment point and serves as the name for the point that can later be accessed by the attachment attributes.

The second way of specifying an attachment point is like the first except that a point number instead of a path number is used to specify the base point. Instead of the `gpath` function, one uses the `gpoint` function. This can only be used if the exact point number is known. It would be useful if the base point was in a path with more than one point and wasn't the first point.

The third way is to specify the base point in terms of x and y coordinates along with optional x and y offsets. This is particularly needed when it is not possible to know the path or point number of the attachment point, or the glyph does not actually contain a real point that attachment can be based on.

```
clsBase { udap = point(advancewidth/2, bb.top, 0, bb.height/10);
          ldap = point(advancewidth/2 , bb.bottom) };
        /* offset not required */
```

Simple integer mathematical expressions (`+`, `-`, `*`, `/`) are allowed, as well as simple functions `min` and `max`, since often locations need to be calculated. Also a C-style conditional statement is possible: `<condition> ? <true-expression> : <false-expression>`. Note that the point names are very much like user-defined variables. Later they will be referenced by slot attributes in the positioning rules.

See the section on positioning and units in the next section for details as to the meaning of the metrics in these specifications. The first attachment point (`udap`) is located at the center of the glyph horizontally and 10% above the top of the bounding box of the character. The second attachment point (`ldap`) is located horizontally centered at the bottom of the bounding box of the glyph.

When a base point is specified in terms of x- and y-coordinates using the third approach above, Graphite, with its affinity for attaching to real design points, will try to locate an actual on-curve point

close to the x- and y-coordinates. If it finds one, it will then associate the base point with the on-curve point and the attachment point will adjust with the hinting of that on-curve point. The directive `PointRadius` gives the default hunting range to find an on-curve point which is considered identical to the base point, so the base point is moved to be the same as that on-curve point. This value defaults to `2m`.

3.5.1.2 Ligatures *[Not implemented]*

True ligatures can be considered to have components which are visible and which may want to be identified within the ligature. In the glyph table we define glyph attributes for the bounding box of each component in the ligature with the `component` (or `comp`) keyword. The correspondences between the ligature components and underlying glyphs are handled in the substitution table when the ligature is substituted for the underlying glyphs.

```
oeLig {component.o = box (0, bb.bottom, advancewidth/2, bb.top);
    comp.e = box (aw/2, bb.bottom, aw, bb.top)};
```

This example introduces the `box` function for defining the bounding boxes in a ligature. It is similar to the functions used for specifying attachment points.

Note: as of Graphite2 version 1.3.12, ligature components are not supported.

3.5.1.3 Directionality

Directionality support is based almost directly upon the directionality description in Unicode. Unicode specifies that a codepoint may take on a number of different directionality types of which only a subset are relevant at the glyph level.

In Graphite, the `directionality` (or `dir`) attribute for a glyph may take on any of these numeric values:

Numeric Value	Unicode Type	GDL Label	Description
0	ON	`DIR_OTHERNEUTRAL`	Other Neutrals (default)
1	L	`DIR_LEFT`	Left to right, strong
2	R	`DIR_RIGHT`	Right to left, strong
3	AR	`DIR_ARABIC`	Arabic Letter, right to left, strong
4	EN	`DIR_EURONUMBER`	European Number, L to R weak
5	ES	`DIR_EUROSEPARATOR`	European Number Separator, L to R weak
6	ET	`DIR_EUROTERMINATOR`	European Number Terminator, L to R weak
7	AN	`DIR_ARABICNUMBER`	Arabic Number, R to L weak
8	CS	`DIR_COMMONSEPARATOR`	Common Number Separator, L to R weak
9	WS	`DIR_WHITESPACE`	Whitespace, neutral
10	BN	`DIR_BOUNDARYNEUTRAL`	Other formatting and control characters (ignored in processing bidirectional text)

Glyphs receive a directionality by virtue of the Unicode codepoints which map to them. Values for unmapped glyphs, pseudo-glyphs, or Private Use Area (PUA) codepoints are defined by setting the `dir` glyph attribute. If a one of these glyphs is not explicitly assigned directionality, it will be considered neutral (ON). The `dir` attribute can also be set in the `substitution` table.

See the Unicode Standard Annex #9 for a full description of the bidirectionality algorithm and the meanings of the above values: www.unicode.org/unicode/reports/tr9.

3.5.1.4 Breakweight

Each glyph also has a `breakweight` (or `break`) attribute which describes whether line-breaking can occur after such a glyph and at what level.

10	white-space break	BREAK_WHITESPACE	30	letter break	BREAK_LETTER
15	word break	BREAK_WORD	40	clip break	BREAK_CLIP
20	intraword break	BREAK_INTRA			

To indicate a possible line-break *before* a glyph, the `break` attribute should be negative (e.g. -10).

Values for unmapped glyphs, pseudo-glyphs, or PUA codepoints are defined by setting the `break` glyph attribute. If a one of these types of glyphs is not explicitly assigned a line-break weight, it will be considered level 30 (letter break). Other glyphs can also be set explicitly, or the compiler will assign a default value. The `breakweight` attribute can also be set in the `linebreak` table.

3.5.1.5 Metrics

Each glyph also has a set of metrics associated with it. See the section on metrics, below, for details on what is available.

3.5.1.6 Mirroring attributes

When the Bidi global variable is turned on, glyphs can set the `mirror.glyph` and `mirror.isEncoded` attributes, which are used for mirroring glyphs in right-to-left scripts. See the Advanced Concepts section for more information.

When Bidi is off, the mirroring attributes are considered undefined and will result in a compilation error.

Note: this feature is only available in the Graphite2 engine.

3.6 Feature Table

Features provide a way to produce rendering variations for a writing system. The `feature` table defines what the features are. Rules within the substitution and positioning tables can then be conditionally executed based on feature settings in the underlying text stream. An application program can determine what the allowable features are and can set them in the text stream it provides to the renderer.

For example, you might create a Graphite renderer with a feature that allows several options with regard to the creation of ligatures. As the calling application passes the Graphite engine a string of text to be rendered, that text contains mark-up indicating which kind of ligature replacement is desired. The Graphite rules are fired conditionally based on that mark-up, so that only the rules appropriate for the kind of ligatures requested will be fired.

Features are similar to glyph and slot attributes in that each glyph in the text to be rendered holds values for each feature, glyph attribute, and slot attribute. Features are different from both glyph and slot attributes in that feature values are determined by the calling application; glyph attributes and slot attributes are entirely private to the Graphite rendering process. While glyph attributes are specified in the glyph table and slot attributes are set by the rules, feature values (at least in the current version of Graphite) cannot be modified by the engine; they are read-only.

Each feature declaration consists of a structure of information regarding naming and possible settings, etc. Features follow a standard variable naming structure whereby `.` is used as a variable structure separator. Thus in the example below, `ligatures.id` may be thought of as the `id` sub-variable of `ligatures`.

Each feature must be declared in the `feature` table. For example, for a feature called “ligatures”, your GDL program might say:

```
table (feature)
ligatures.id = 345;
ligatures.name.LG_USENG = string("Ligature Replacement");
ligatures.default = std;
ligatures.settings.all.value = 3;
ligatures.settings.all.name.LG_USENG = string("All");
ligatures.settings.std.value = 2;
ligatures.settings.std.name.LG_USENG = string("Standard");
ligatures.settings.min.value = 1;
ligatures.settings.min.name.LG_USENG = string("Minimal");
ligatures.settings.no.value = 0;
ligatures.settings.no.name.LG_USENG = string("None");
endtable
```

Notice that there are four possible settings for this feature, the default being “standard.”

This example introduces the `string` function which returns a list of Unicode values, one for each character in the string. It accepts standard C character escape codes (`\t`, `\n`, `\\`, etc.). The `string` function is much like the `codepoint` function in that it also takes an optional codepage parameter. Thus we could have written:

```
ligatures.name.LG_USENG = string("Ligature Replacement", 1252)
```

This information can also be expressed hierarchically as part of the feature definition. Thus our example would become:

```
table (feature)
ligatures {
   id = 345;
   name.LG_USENG = string("Ligature Replacement");
   default = opt;
   settings {
      all {
         value = 3;
         name.LG_USENG = string("All");
         }
      std {
         value = 2;
         name.LG_USENG = string("Standard");
         }
      min {
         value = 1;
         name.LG_USENG = string("Minimal");
         }
      no {
         value = 0;
         name.LG_USENG = string("None");
         }
      }
   }
endtable
```

The `id` is used by applications so that they can store a language independent reference to a feature without having to go through the language system and full names, which may vary.

Features and feature settings each have a `name` element which is language specific. The fall-back language is `LG_USENG`. The `default` element is set to the identifier or value of a `setting` element and is the default setting of the feature if no setting is applied. If there is no `default` element, then the setting with the lowest value is chosen.

A feature need not have settings. A feature with no settings specified results in a boolean type feature. It can have two possible settings: `1` (`true`) and `0` (`false`) .

3.6.1 Styles *[Not implemented]*

One special use of features is to support font styles such as bold and italic. There is a special feature ID which is used to handle all styles. By defining a feature with this ID, one is specifying that this feature will be tested against the style information available for a text run. There is also a pre-defined set of possible settings which correspond to font styles.

```
table (feature);
style {
   id = STYLE_GENERAL;
   name.LG_USENG = string("style");
   settings {
      regular {
         value = STYLE_REGULAR;
         name.LG_USENG = string("regular");
         }
      bold {
         value = STYLE_BOLD;
         name.LG_USENG = string("bold");
         }
      }
   }
endtable;
```

Note: as of Graphite 2 version 1.3.12, this feature is not supported.

3.6.2 Language Table

The language table can be used in conjunction with the feature table to define sets of features that are associated with a given language. Languages are identified in terms of ISO-639-3 identifiers. Groups of languages can be assigned default feature values. The syntax for the language table is:

```
table (language);

language-group {
   languages = ( ISO-ID, ISO-ID, … );
   feature-name = value;
   feature-name = value;
   etc.
};

language-group { … };

endtable;
```

The features listed must be those defined in the feature table, and therefore the language table must follow the feature table in the file structure. The language group label can be any arbitrary string that meaningfully describes the group of languages.

An example of language definitions for Arabic script might be:

```
table (language);
```

```
sindhi {
   languages = ( "snd", "sd" );
   meemAlt = sindhi;
   easternDigits = sindhi;
   shaddaKasra = sindhiUrdu;
};

kurdish {
   languages = ( "bdh", "ckb", "kmr", "kur", "sdh" );
   hehAlt = kurdish;
};

urdu {
   languages = ( "urd", "ur" );
   hehAlt = urdu;
   easternDigits = urdu;
   shaddaKasra = sindhiUrdu;
};

endtable;
```

3.7 Name Table

The name table is used to insert multilingual text into the compiled GDL file. Each compiled file has a section referred to as the name table where these strings are stored. For those familiar with TrueType, this is an extension of that standard name table.

```
table (name) {CodePage = 1252};
NAME_AUTHOR.LG_USENG = string ("John Quick");
endtable;
```

This would insert the Unicode string corresponding to "John Quick" in the compiled GDL 's name table. The string will have a language ID of LG_USENG and a name ID of NAME_AUTHOR. The language IDs and some name IDs have been standardized with a semantic meaning by the computer industry for use in a TrueType font's name table. Many of these are accessible as #defines from a standard #include file as shown by LG_USENG in the above example. In addition Graphite has defined additional name IDs that may be useful as shown by the NAME_AUTHOR label.

Users can also use arbitrary integers to create their own name IDs provided there is no collision with a standard ID or with an ID that Graphite uses. Specifically, to be safe users should only use values greater than 40960 (0xA000) for their name IDs. Of course, only the user will know the semantics of their name IDs, unlike the well-known semantic meaning of standardized IDs.

3.8 Global State Variables

There are several global state variables that are used for justification. They are described in the Advanced Concepts section.

4 Data Processing

In processing a run of text, Graphite takes the text through a number of processes in order. The text starts off as a series of Unicode codepoints and ends up as a sequence of positioned glyphs. The processes, in order, are:

- Convert Unicode characters to glyphs
- Set line-break values
- Apply substitution rules
- Internally reorder mixed direction text
- Apply positioning rules
- Perform final placement

(For simplicity, the steps above do not reflect justification. See the Advanced Concepts section for a discussion of how justification affects the Graphite processing model.)

This section takes each process in order. But first, some further details on how rules interact with each other.

4.1 Processing Rules

Rules do not exist in isolation, and here we consider their interaction. In what order are rules tested and executed? What happens after a rule matches?

For the most part, the processing model should be sufficiently intuitive that it can be ignored. But there are times when an author may need to consider rule interaction, and then the processing model becomes highly significant.

4.1.1 Scan Position

Rule matching can be considered as having a scan position, the current location in the input stream. When matching a rule, the scan position corresponds to the first underscore in the context (or the first item in the rule if there is no context). When a match occurs, the action (the part to the right of the >) is performed, and the scan position is moved to just after the last underscore in the context (or after the last item in the rule if there is no context). The new scan position is then used to start searching for new rules. If no rule is found, the scan position is advanced by one glyph and the process restarted.

Why is the scan position moved to such a strange location—after the last underscore in the context?

Consider the following rule:

```
X > Y / A _ A
```

with the following input:

```
A X A X A X A
```

In order for the output we might expect (`A Y A Y A Y A`), it is necessary for the processor to rescan the `A` that occurs at the end of the context. This is achieved by placing the scan position just before the final `A` in the context. More generally, the solution is to place the scan position after the final _ in the context.

For rules with no left-hand side, scan position is also adjusted to be after the final _ in the context, or after the final glyph in the rhs. However, this may not be the most convenient approach, particularly for positioning rules. For instance, it may be useful to only advance by one glyph at a time, so that a sequence of diacritics of indeterminate length may be stacked on top of each other without reference to the base character at each step:

```
clsBase clsDiacritic {attach {to = @1; at = udap; with = lap}};
clsDiacritic clsDiacritic {attach {to = @1; at = udap; with = lap}};
```

Always adjusting the scan position past the last glyph would preclude this set of rules; several passes would be required to produce the desired result. Instead, the context can include the ^ symbol as an indicator of where the scan position should be placed after the rule is applied:

```
clsBase clsDiacritic {attach {to = @1; at = udap; with = lap}}
      / _ ^ _ ;
clsDiacritic clsDiacritic {attach {to = @1; at = udap; with = lap}}
      / _ ^ _ ;
```

This says that the new scan position should be after the first glyph in the context, rather than in the default position after the second. This allows rescanning of the second item in the rule.

Note, however, that this feature makes it possible to write rules that result in infinite loops during rule application, that is, where a sequence of rules never allows the scan position to advance at all for a given input. In fact, it is quite easy to write such rules, and even with care it may be possible to get this sort of nasty effect. As a safety net, therefore, a directive exists indicating the maximum number of rules that can be applied without the scan position advancing. If this number is reached, the scan position is forcibly advanced before the next rule is applied. The default for this variable is 5; it can be changed using the `MaxRuleLoop` directive:

```
pass (2) {MaxRuleLoop = 10};
   /* … */
endpass;
```

Typically this directive is used on a pass, but it can be used in any environment.

4.1.2 Features

The feature mechanism provides a way for users to parameterize rendering. Thus users can change the style of a rendering according to their preferences. For example, they might want to enable or turn off ligature substitution, or even switch between a script and its transliteration. The feature mechanism interacts with an application's user interface to allow a user to set different features for a run of text to change how it is rendered. There is also a mechanism to map font styles within a text run to features, thus allowing for features to be used with styles also.

Features appear within a rendering description by specifying what rules are available for matching, via *feature constraints*. These feature constraints can be tested either for a set of rules using the `if` statement or for specific slots using tests within the context of a rule. The former method uses feature constraints that look much like `if()` statements in C. They can be single-line or multi-line and can be nested. The. `else` statement is also supported. An `endif` statement is required following the conditional rules. A terminating semi-colon is optional.

Within the test for either method, all the usual C logical operators can be used (`&&, ||, ==, !=, !, >, <, >=, <=`) along with parentheses for subexpression grouping. The order of precedence for these operators is as in C. Features allow the end user to control the way in which Graphite renders the underlying data. By setting different features to different values, it is possible to completely change the way a piece of text is rendered. A typical feature test might be structured as:

```
if (ligatures == no)
   // rule
   // rule
else if (ligatures == all) // “else if” is available this way,
                      // if both are on the same line
```

```
    // rule
    // rule
endif;
```

For a rule from above to be applied, the feature must be true for all the slots in a rule. To test specific slots, the test can be specified within the context of a rule.

```
gA  >  gB  /  _ {ligatures == all};
```

Slot tests can only be used in rules with an explicit context.

4.1.3 Slot Constraints

In addition to testing for features in the context of a rule, one can also test any readable attribute. This mechanism allows for constraints to include glyph attributes, glyph metrics, and slot attributes. These attributes cannot be tested using an `if` statement.

```
gA > gB / _ {bb.height > 1000m};
```

4.1.4 Bidirectionality

4.1.4.1 Rule Item Order

When Graphite processes a rule by matching glyphs to the elements of the rule, it always works in logical order rather than physical order (with one exception that we'll discuss below). At the same time, the elements in a rule are always logically ordered from left to right (assuming, of course, a left-to-right editor in which one is writing GDL code!).

```
item1 item2 item3 > replacement1 repl2 repl3
   / context1 _ _ _ context5;
```

This means that for a right-to-left writing system, the items in the rules are written in GDL in the opposite order from the way they are ultimately displayed to the user. For instance, suppose you have a rule matching glyphs A, B, and C, in that order:

```
gA gB gC {...};
```

In a right-to-left writing system the final output would be "CBA" but the rule is still written as above.

4.1.4.2 Internal and Final Reordering

In the linebreak and substitution tables, the order of the items in the rules always corresponds to the underlying text order. For instance, if you have a right-to-left writing system and your underlying text is "ABC 123 DEF", this is the order of the glyphs during these two tables.

Some writing systems have *internal* bidirectionality, that is, there are sequences of glyphs that are in the opposite direction from the overall flow of the text. This reordering occurs at end of the substitution table, in a special pass called the "bidi pass", just before the positioning table.

So if, in our right-to-left writing system, numbers are written left to right, the bidi pass will change our sample underlying order from "ABC 123 DEF" to "ABC 321 DEF". This is the order that is used during the positioning passes.

At the very end of the positioning passes, just before glyphs are displayed on the screen, *final* reordering is done for right-to-left writing systems. Our final example output becomes "FED 123 CBA" and this is how it is displayed.

4.1.4.2.1 *Contextualization Across Direction Boundaries*

Note that the above process makes it possible to do contextualizations across internal direction changes using *logical* adjacencies in the substitution table, but using *physical* adjacencies in the positioning table.

For instance, as part of the substitution table we could write a rule recognizing the logical adjacency of the "C" and the "1" in our example text, but not the physical adjacency of the "C" and the "3". On the other hand, the positioning table could contain a rule recognizing the physical adjacency of the "C" and the "3", but not the logical adjacency.

4.1.5 Tables

While syntactically two tables may be interleaved, in terms of data processing, the elements of a table are all grouped together. The substitution and position tables are strictly ordered: all the rules from the substitution table are applied before any from the position table are applied. We will examine these tables in turn in following sections.

Certain of the slot attributes are associated with certain tables. If a slot attribute is set in a table that does not recognize it, a compile-time warning will be given and the setting will be ignored.

Each table is made up of one or more passes. The passes, in turn, contain the rules.

4.1.6 Passes

Not being able to reprocess data that has been output can be a big problem when there is complex processing involved. For this reason a table may allow a multi-pass processing model. This allows the GDL file author to have one set of rules, which are considered to run together, once, over the glyph string. They can then have another set which are run once over the resulting string, and so on for as many passes as they need.[1]

`Pass()` statements act somewhat like `if()` statements. They can be single- or multi-line. Each pass has a number and the data stream is processed in the order of the passes. It is not necessary to group all rules associated with one pass together. This allows for rules to be grouped according to other criteria such as linguistic structure. You might use a `#define` to name the pass numbers rather than embedding their numbers in the code. This has the advantage of making it easier to insert new passes without having to change every `pass()` statement.

If no pass statements have been encountered for a table type, the current pass is 1. The current pass for a given table type is remembered, so when the table type changes, the current pass number for the new table is used.

A typical structure for a multi-pass description might be:

```
pass(1);
   /* rules */
if (feature == yes)
   /* rules */
endif;
   /* rules */
endpass; // pass 1

pass(2);
if (feature == yes)
   /* rules */
else
   /* rules */
endif;
endpass; // pass 2
```

[1] There is one implementation difficulty which it would be worth bearing in mind if it is intended for the description to be used to generate GX tables. The GX `mort` table is where the substitutions are made. The problem with the `mort` table is that any insertions cannot be used in any subsequent passes over the data. Therefore insertion rules should only be used in the final substitution pass of a multi-pass description.

Semi-colons following the `pass` and `endpass` statements are optional.

4.1.7 Ordering of Rules

Within a pass, rules are given a priority order. This priority is based on the length of the context; i.e., longer matches have priority over shorter matches. Thus:

```
A B C > X Y Z / _ _ _;  // priority 1
A > D / _ B;       // priority 2
```

will be tested in the order given here regardless of the order in which the rules appear in the file. If several rules have the same length, they will be processed in the order they appear in the file. This allows the GDL author to control these subtle ordering issues.

For the most part, ordering should not need to be an issue of consideration for a GDL author. The priority-based-on-rule-length approach used by Graphite has been found to be the most natural in other rule-based systems.

Linebreak pseudo-characters in the context are counted for ordering. Inserted glyphs are not counted when determining rule ordering. See below for these topics.

4.1.7.1 Optionality

When calculating the length of a rule, it should be noted that rules are not dealing with strings, as such, but with a list of glyphs. When a rule contains an optional element, it is internally resolved down to two rules: one with the element and one without. These resolved rules are then inserted at their appropriate locations in the priority order.

For example:

```
B  >  Y  /  W  A?  _  C;
C  >  D  /  B  _  E;
```

An input sequence of “WABC” will result in “WAYC” as might be expected. But what about “WBCE”? The above rule set will result in “WYCE”[1] because although the first line generates a rule that has only 3 items (when the optional item is ignored), this shorter rule is still the same length as the second and precedes it in the file, and therefore it takes precedence over the second rule.

4.1.7.2 Rules beginning with context items

The scan position is considered to be before the first item in the left-hand side, that is, just before the first underscore in the context if any. (Note that this represents a change from earlier versions of Graphite.) Therefore, given the following rules:

```
A  >  B  /  W  _;
A  >  C  /  _  X  Y;
```

and the input “WAXY”, the output would be “WCXY”. Neither rule is considered to match when the scan position is at the beginning of the input, so the W is simply copied to the output stream with no change. Now the scan position is before the A, and both rules match. But the second rule has precedence due to its longer context. It fires, and the first rule is ignored.

[1] For this particular problem, an alternative solution might be to argue that all of the context before the first match character (in the case of the second rule, the match character is `C` and this pre-context is `B`.) should be ignored as part of the length as per `pre()` in CC. This would result in the same solution but would weight rules towards those with earlier information on the left hand side of the rule.

4.2 Converting Characters to Glyphs

Without smart font capabilities, the process of converting a Unicode string to the corresponding glyph string would consist of looking each Unicode codepoint up in the `cmap` table in the TrueType font, where it would find the corresponding glyph number.

For the most part, this is what Graphite does also, and in most situations GDL authors can leave the first phase of conversion as just that. In the case of two or more Unicode codepoints mapping to the same glyph, Graphite, by default, treats these as separate glyphs. This is achieved using the *pseudo-glyph* mechanism, which is discussed, in the Advanced Concepts section. It is also possible to define one's own pseudo-glyphs.

4.3 Linebreak

One of the Graphite engine's functions is to produce line-breaks in cooperation with the calling application. As the application asks for a range of text to be rendered, it specifies how much physical space is available, and the engine produces an appropriate break point in the text.

Graphite's line-breaking algorithm is based on the `breakweight(`or `break`) attribute of a slot. The `break` attribute indicates the level of appropriateness for a line-break at that point in the text. Graphite can set a line-breaking weight either in the `glyph` table or `linebreak` (or `lb`) table.

The `lb` table is made of rules that set line-breaking weights for slots which override any line-break weights set in the glyph table. These rules look very much like position rules, but they only set the `break` slot attribute. As potential line-breaks are determined, these weights are considered as *preferences* or hints for where a line-break might occur. Nothing in GDL can explicitly *force* a line-break at a specific slot; the linebreak table only suggests possible breakpoints.

See section 3.5.1.4 above for a list of possible breakweight values.

Since the `linebreak` table cannot perform substitutions, it is an error to include a rule with a lhs.

4.4 Substitution

This is the meat of Graphite. In addition to simple replacement, as has been covered in the previous section, there are more complex tasks that substitution rules can be used for.

The substitution phase runs the substitution rules, which can replace, reorder, insert and delete glyphs in order to get the right glyphs into the right order in the glyph string. In some scripts, there is hardly any substitution work to be done. In others, the substitution rules become multi-pass and highly complex. The Graphite language aims to provide the expressive power needed to meet the most demanding of needs.

Substitution rules can be thought of as transforming an underlying glyph sequence into a surface glyph sequence. In a multi-pass system, each pass does one set of transformations, and the "surface" glyphs for one pass are the "underlying" glyphs for the next.

The substitution table consists of a multi-pass set of rules. The table is identified by:

```
table(substitution);
```

4.4.1 Selecting Glyphs From A Class

Whenever a class is used in the right hand side of a rule, Graphite must select one glyph from the class when applying the rule. Graphite selects a glyph from a right hand class by its position within the corresponding element's class in the left hand side of the rule.

```
c(192) > clsLowTones$1 / clsVowel _ ;
```

In this example, the glyph corresponding to the 8-bit codepoint 192 is to be replaced by a glyph from the class `clsLowTones` which has a correlation with the `clsVowel` class. The `$` marks the following number as identifying which element in the rule should be used to index this array.

Notice that the number is with respect to the context and not to the list of substituted glyphs. This is *always* the case. *All numbers which refer to a slot in a rule are with respect to the context* and not to the left hand side of the rule.

The $ symbol differs significantly from the `@` symbol. While the `@` symbol refers to the glyph in a neighboring slot, the $ is used to access the *index* of the glyph in the specified class. Consider for instance, the following rule which is similar to the one above:

```
clsVowel c(192) > @1 clsLowTones$1;
```

In this rule, the `@1` has the effect of replacing the glyph in slot 1 with the glyph in slot 1—in other words, leaving it unchanged. The second item in the rule places an element from the clsLowTones class—not the clsVowel class—in slot 2. The purpose of the `$1` is to select the glyph from clsLowTones the corresponds to the glyph in slot 1, based on the latter's index in the clsVowel class.

4.4.1.1 Slot aliases

In a very long rule, it may be error-prone to actually use numbers as glyph selectors. For that reason, it is possible to define a temporary alias to be associated with a slot, and use that name to indicate the position. So the above example might also be written:

```
c(192) > clsLowTones$vowel / clsVowel=vowel _ ;
```

Aliases may also be placed in the left- and right-hand sides of the rule. They may not begin with a number.

4.4.2 Reordering

Another action performed by substitution rules is reordering. The right-hand side specifies this with the `@` symbol and a number. The number is chosen as in selection. For example:

```
clsCons clsVowel1 clsVowel2 > @2 @1 @3
```

This places the glyph matched by clsVowel1 first followed by the glyphs matched by clsCons and clsVowel2. Slot aliases can be used instead of numbers. See the Associations section for how the reordered glyphs are associated with underlying glyphs.

4.4.3 Associations

One of the most complex areas of script description is that of associations. In order for the cursor to behave correctly and to reflect either the underlying form to the surface or the surface to its underlying form, it is necessary to associate surface glyphs with their underlying characters. For the most part, this association is done automatically, but there are occasions where a more complex relationship is required.

Graphite provides a rich mechanism for associating surface and underlying characters. In fact, the model which Graphite uses is more complicated than that presented here, and users who need an advanced understanding of this area should read the section on Cursor Hitting in the Advanced Concepts section later in this document.

In addition to being able to work relative to another glyph, it is also necessary to identify which glyph a substitution is associated with. Consider the following (slightly false) example from Devanagari:

```
clsCons gDepI > gDepI$2 clsCons$1;
```

In Devanagari, the letter I is rendered before the consonant it is stored after. This example would achieve the task of re-ordering the two glyphs, but there remains the question of what happens to the

cursor. If we were to select the initial `gDepI` on the surface, and delete it, we would in fact be deleting the consonant in the underlying text. What is needed is some way to indicate that the `gDepI` on the surface is actually the same as the `depI` underlying it. The colon (:) is used to create this association:

```
clsCons gDepI > @2:2 @1:1;
```

The `@2:2` specifies that the consonant should be replaced by the second glyph (`@2`) and should be associated with the second glyph (`:2`). Here the `:` is used to indicate which underlying glyph a surface glyph is to be associated with.

Unfortunately, this is a rather cumbersome way of describing what we want. In order to simplify the syntax for simple situations, the following equivalencies have been set up:

`@2` is equivalent to `@2:2` which is equivalent to `@:2`

Referencing a glyph assumes an association with it, and associating with a glyph assumes a reference to it. In effect, there is only a need to give both numbers if they are different. We can now write out re-ordering rule as either of:

```
clsCons gDepI > @2 @1;
clsCons gDepI > @:2 @:1;
```

although the first syntax is preferred.

It is also possible to do re-ordering and substitution at the same time.

```
clsVowel clsTone? > clsUpperTone$3:3 clsUpperVowel$2:2 / clsCons _ _ ;
```

This rule selects the uppercase version of the vowel that was in the input (`clsUpperVowel$2`), and uppercase version of the tone that was in the input (`clsUpperTone$2`). It also reorders the two so that the tone is rendered before the vowel, but is properly associated with the original tone in the input (`:3`), while the vowel is associated with the original vowel (`:2`).

Notice that the numbers used for `$` here are not 1 and 2 as might be expected if the `$` were referring to an element in the left-hand side of the rule, but 2 and 3 which are elements in the context. The numbers for `$`, `@` and `:` always refers to elements in the context and their position within the context. Notice also that the `$` operator does not change association, thus without the `:` in the above rule the glyphs would not be properly reordered.

It is helpful to keep in mind the distinctions between the colon(:), @,and $:

- The colon (:) is to used to specify an association between slots in the input and slots in the output—and ultimately between underlying characters and surface glyphs.
- The @ refers to the glyph that is in the specified slot, *and* has the effect of creating an association (that is, a colon is implied if none is present).
- The $ is used to select a glyph to put into the output based on the index of a corresponding glyph in the input. In other words, it creates a mapping between the members of two classes. The $ has no effect on associations—the mapping between underlying characters and surface glyphs.

Slot aliases may be used in place of the numbers:

```
clsVowel=V clsTone=T? > clsUpperTone$T:T clsUpperVowel$V:V
   / clsCons _ _ ;
```

4.4.4 Insertion & Deletion

An alternative way of dealing with our dependent I is to delete the `I` from its place after the consonant and to insert it before the syllable. This would be done by the following rule:

```
_ gDepI > @3 _ / _ clsConsC _ ;
```

This rule moves the dependent I to the gap before the consonant. Notice how the _ is used on the lhs to indicate an insertion, while on the rhs it indicates a deletion. In our example above, we deleted `gDepI` from slot 3 and inserted it in slot 1.

Insertion and deletion create some special problems that require associations to be specified. Consider another example:

```
ZWJ clsCons > _ clsConsJoin / clsCons[1] _ _ ;
```

If a glyph is deleted, the question remains as to what to do with the cursor. If the cursor is placed between the `clsCons` and the `clsConsJoin`, where should it be placed in the underlying text: before the `ZWJ` or after it? It is necessary to indicate which surface glyph a deleted glyph is associated with. This is done by allowing a surface glyph to be associated with more than one underlying glyph. In the above example, we may want the `ZWJ` to be associated with the following consonant. Thus our rule should be written:

```
ZWJ clsCons > _ clsConsJoin:(2 3) / clsCons _ _ ;
```

This indicates that the `clsConsJoin` is associated with both the `ZWJ` and the `clsCons`. Notice again how the numbers refer to elements in the context rather than on the right hand side. This is to allow linking to elements in the context which are not replaced by the rule. For details of what happens if no associations are made for a deleted glyph, the reader is referred to the Advanced Concepts section on cursor hitting. The aim is that not specifying an association for a deleted glyph should result in the most natural behavior occurring. In ambiguous cases, this behavior may be wrong for your requirement. Therefore, *associations should always be specified for deleted glyphs*.[2]

Insertions are also possible. In Thai script, amongst others, some vowels are split up and consist of a number of glyphs arrayed around a base consonant. The following is an example of one of them:

```
_ VSchwa > VE:4 VShort  /  _  clsCons clsTone?  _ ;
```

This inserts the two parts of a vowel diacritic (before and after the consonant and tone) and associates them both with the `VSchwa`. Again, as for deleted glyphs, unless there is only one non-null element in the left hand side of the rule, *associations should always be specified for inserted glyphs*.

Inserted slots are not counted when determining rule precedence.

4.4.5 Ligatures

One use of attributes in the substitution table is to associate underlying cursor positions with surface cursor placement points in a ligature. Since the relationship between an underlying form and the perceived components of a ligature is not necessarily one-to-one in a particular context, it is necessary to indicate the relationship. This is done by associating each of the possible cursor locations in a ligature with a position in the underlying string.

Consider the following example:

```
Co ZWJ Ce > Coe:(1 2 3) _ _ ;
```

The relationship between the cursor position, which is available in the œ between the o and e, and the underlying text must be marked somehow. We would probably want the cursor to be placed between the `Co` and the `ZWJ`, since the `ZWJ` is really modifying the `Ce`.

The ligature component association is indicated using glyph attributes previously defined on the ligature:

---

[1] This is probably not the way that conjuncts would be handled.

[2] I assume that not associating can be made an error condition, but there may be something more useful we can do.

```
Co ZWJ Ce > Coe:(1 2 3){component{o.reference = @1; e.ref = @3}} _ _ ;
```

This indicates that the whole ligature is associated with all three underlying codes. (Notice the glyph deletion to make the ligature.) Each component is named with a glyph attribute and has a `reference` (or `ref`) attribute which refers to the underlying glyph associated with it. The named components correspond to bounding boxes which are defined with glyph attributes for the sub-regions of the ligature associated with each component.

Note: as of Graphite2 version 1.3.12, ligature components are not supported.

4.4.6 Line-break Pseudo-glyph

While most control characters are dealt with at the application level, one important pseudo-glyph is kept in the glyph stream. This is a glyph to mark the end of a line, either the start or the beginning. This allows line-based context substitutions. For example:

```
clsCaps > clsSwashCaps / # _ ;
```

The line-break glyph is identified using the `#` character and may only appear in the context of a rule. It is counted when determining rule order and when determining slot numbers for references (with `$`, `:`, and @). It cannot be optional, and it is not permissible to reference the slot it occupies.

See the section on the `linebreak` table for more information on how line-breaking is done.

Each line-break pseudo-glyph has a break weight associated with it. This break weight can be determined by asking for the `breakweight` (or `break`) slot attribute. For instance, you can use the `breakweight` attribute to determine whether to insert a hyphen at a break:

```
_  >  gHyphen /  _  # {breakweight == 2};
```

As mentioned before, the `break` attribute is also used to specify line-break preferences for glyphs or slots. The usage in the above rule of the same attribute differs in that one is testing the actual type of break that resulted from the line-breaking process.

The meaning of the values for the `breakweight` are the same in both usages: `BREAK_WORD`, `BREAK_INTRAWORD`, `BREAK_LETTER`, and `BREAK_CLIP`.

4.5 Directionality

The substitution process does not have the sole responsibility for getting all the glyphs into the right order in the output glyph string. There is also the Unicode directionality property to take into consideration. This takes into account that, for example, in a right-to-left script, European numbers may be read left-to-right.

Rather than leaving this to the substitution table, an extra process is inserted which takes the Unicode directionality properties of each glyph and from these, does further reordering to get the final glyph order. Between the substitution rules and the positioning rules, any glyphs which have a direction opposite of the overall writing system direction are reversed. This process is called internal reordering.

Glyphs all receive a directionality by virtue of the Unicode codepoints which map to them. Values for unmapped glyphs, pseudo-glyphs, or PUA codepoints are defined by setting the `directionality` (or `dir`) attribute in either the `glyph` or `substitution` table.

Based on the `dir` attribute for each glyph, the glyphs are reordered according to their directionality and the Unicode bidirectional algorithm. Rules are always written based on the underlying text order except for positioning rules which must take into account internal reordering.

If you are working with a script that you are sure has no internal bidirectionality, you can set the `Bidi` global to false. This is an optimization that allows the rendering engine to avoid the superfluous step of performing internal reordering.

4.6 Positioning

Once all the glyphs are in the right order for output, we can go about positioning them. The glyph order for positioning is the same as for the underlying codes. Thus, if the primary direction of the text for this run is right-to-left then moving forwards through the glyph stream moves us left.

The positioning table consists of a multi-pass set of rules. The table is identified by:

```
table(positioning)
```

The primary mechanism provided for glyph positioning is attachment. It is possible to define attachment points on glyphs (as glyph attributes). These attachment points may then be used to position two glyphs with respect to each other using the `attach` (or `att`) slot attributes. A typical rule used for diacritic attachment might be:

```
clsBase clsDia {attach.to = @1; attach.at = dap; attach.with = base};
```

The strings `dap` and `base` refer to the named points which are glyph attributes.

Due to the nature of attachment, it is an error to attach to glyphs that are not visually adjacent. In other words, attachment must be done between two glyphs which are adjacent or between two glyphs which are separated only by glyphs which are attached. (This may be difficult or too costly for the compiler to check for.)

Note that it is an error for a rule in the positioning table to have a lhs.

4.6.1 Shifting

In addition to being able to attach glyphs to other glyphs, there is the ability to *shift* a glyph whereby it is moved (along with all its dependent attachments: those glyphs attached to it). For such purposes we use the `shift` slot attributes. `Shift` is the offset from a glyph's normal placement (after attachment is processed).

```
clsBase clsDia {shift.x = -@1.advance.x/2;
          shift.y = diaheight};
```

This would shift the diacritic above the base glyph. Shifting does not change the screen position of the following glyphs. Attaching a diacritic does not alter the slot's `shift` value.

In left-to-right fonts, shifting by a positive number moves the glyph to the right, while a negative number moves the glyph to the left. In a right-to-left font, the opposite is true. In other words, shifting by a positive amount always moves the glyph “further along” in the direction of the script’s orientation.

To aid in shifting and other positioning operations, it is possible to interrogate a glyph for the value of one of its attributes. This is done by using a dot notation and the name of the metric needed (as above). A full set of mathematical operators is available for calculations (`+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`, `min`, and `max`).

4.6.2 Advancing and kerning

To alter the screen position of following glyphs the *advance* needs to be modified. There are two `advance` (`adv`) slot attributes describing the distance between the origins of two glyphs. The glyph metrics `advancewidth` (`aw`) and `advanceheight` (`ah`) also exist. The default `advance` value is the advance of the glyph; `advance.x` defaults to `aw`, `advance.y` to `ah`.

```
gOverhanger {advance.x += overhang} / _ clsAny;
```

This example would move all glyphs on the line following gOverhanger to the right by the overhang amount. `Overhang` is just a named glyph attribute. (Note that `clsAny` is not a special class. It was created by the author using normal class definition.) It is possible to set `advance` with = instead of += to ignore the glyph advance metric.

Note that, like shift, the meaning of the advance value is determined by the direction of the font. A positive advance value moves the following glyph "further along" in the direction in which the glyphs are being laid out. So in a right-to-left font, a positive advance value would cause the following glyph to be positioned further to left than normally.

It would be relatively rare to use both horizontal and vertical advance attributes in a single font. Normally `advance.x` and `advancewidth` are significant only for horizontal scripts, while `advance.y` and `advanceheight` are needed for vertical scripts. An exception might be in a font for a Nastaliq-style (sloping) Arabic script, where the vertical position must be continuously adjusted along with the horizontal advance.

Normally `advance` is used in conjunction with `shift` to accomplish *kerning*. When a glyph is kerned both its screen position and the screen position of following glyphs on the line are moved.

```
gA gW {shift.x = -10m; adv.x = advancewidth - 10m};
```

Since this is such a common operation, the `kern` slot attribute is available. It is implemented by specifying both `shift` and `advance`. `Kern` is not a readable slot attribute; it can only be written. The above rule could be written more simply as:

```
gA gW {kern.x = -10m};
```

In addition to general shift and advance values, it may be that, at a later date, device-specific values or those associating with a control point on the glyph outline may be added.

4.6.3 Composite Metrics

Once two or more glyphs are attached, composite metrics exist for the glyph cluster. During line layout these composite metrics will be used.

```
gLowerI gTilde {attach.to = @1; attach.at = udap; attach.with = bap};
```

In this example, which places a tilde over a lower case i, the need for composite metrics is evident. If only the metrics for the i were used, characters on either side of the i tilde glyph cluster would collide with the tilde. The line layout must be adjusted by using composite metrics derived from the tilde and the i.

These composite metrics can also be accessed in a rule. For example:

```
pass (1);
   gOne gTwo {attach {to = @1; at = dap; with = base}};
endpass;
pass (2);
   gOne gTwo gThree {kern.x -= @1.bb.width.1 / 10};
endpass;
```

The trailing '1' indicates that the composite metrics should be used. Without a number, the metrics for the single glyph in the slot would be accessed.

The above represents a simple case of a more general mechanism. There are cases where multiple levels of attachment are needed. A sequence of base characters may have to be attached in a cursive script. Some of those base characters may then have diacritics attached. The diacritics may have other diacritics stacked with them. To keep the various levels organized the `attach.level` attribute is used.

```
pass (1);
   gDia {attach {to = @1; at = dp; with = bp; level = 1}}
      / gBaseOne _;
endpass;
pass (2);
   gBaseTwo {att {to = @1; at = trail; with = lead; level = 2}}
      / gBaseOne gDia _;
endpass;
```

The composite metrics can then be accessed using the level numbers. Continuing the above example, in pass 3:

```
gBaseThree {kern = @1.advancewidth} / gBaseOne gDia gBaseTwo _;       // line 0
gBaseThree {kern = @1.advancewidth.1} / gBaseOne gDia gBaseTwo _;     // line 1
gBaseThree {kern = @1.advancewidth.2} / gBaseOne gDia gBaseTwo _;     // line 2
```

Line 0 would access the metrics for the glyph only. Line 1 would access the composite metrics for the base glyph with its attached diacritic but not with the second attached base glyph. Line 2 would access the composite metrics for all three attached glyphs. Notice how higher numbered levels incorporate the metrics of lower levels. If an attachment is made without a `level` attribute, level 1 is the default.

Sometimes it is desirable to attach glyphs without moving them from their normal positions. The author may want to obtain the metrics for a sequence of glyphs even though they are not visually attached to one another. This can be done using attachment without specifying the `attach.at` and `attach.with` attributes. The `attach.level` attribute can still be used in this case.

Note that composites are only available for glyph metrics, not for normal glyph attributes or slot attributes.

4.6.4 Position

The `position` (or `pos`) slot attribute allows one to determine the distance between two glyphs. It is readable only. `Pos.x` and `pos.y` both exist. `Pos.y` provides the distance of a glyph's upper left corner from the baseline. A single `pos.x` value is not meaningful in and of itself. It is only useful when comparing with or calculating the difference from a second `pos.x` value.

4.6.5 Cursor Placement: the `insert` attribute

One of the difficulties with glyph positioning is working out what the cursor is going to do. We generally have an implicit assumption as to what we want the cursor to do in a particular situation, the difficulty is formalizing this behavior in a way which is both natural and right most of the time.

If two glyphs are attached, it is probably desirable that the cursor not be allowed to come between them. The `insert` slot attribute is used to control this. Normally `insert` is set to true (1) . To prohibit cursor placement before a slot, `insert` should be set to false (0) . The `insert` attribute is automatically set to false when `attach.to` is used, though this can be overridden with a slot attribute setting. `Insert` can also be used in the substitution table, where it may be particularly useful when glyphs are inserted.

Note that in the Graphite system, insertion points and range selections are always defined in terms of the underlying characters, not the rendered glyphs. This means that in practical terms the `insert` attribute applies not to the glyph itself, but to the corresponding character (more specifically, the *first* corresponding character). That is, when the `insert` attribute is set to false on a *glyph*, it indicates that no insertion is permitted before the corresponding *character* in the underlying data.

This is important to keep in mind when reordering is occuring in the data. Consider this example:

underlying data:	A	**B**	**C**	**D**	E
surface glyphs:	a	**c**	**d**	**b**	e

It may be tempting to set `insert` to false on the glyph “b” to prevent insertion in the middle of the reordered cluster (BCD).This will not have the desired effect; instead it will prohibit insertion between the A and the B in the underlying data. A more appropriate action is to set `insert` to false for the glyphs “c” and “d”.

Furthermore, when attachments are involved, the automatic setting of the `insert` attribute can have an unexpected effect. In the following example, suppose that “b” has been attached to “c”.

underlying data:	A	B	C	D
surface glyphs:	a	c	b	d

This has the effect of automatically preventing insertion before the “b”, that is, the B character. But as explained above, this does not have the desired effect of preventing insertion between the attached glyphs (instead it prevents insertion between the A and the B). In this sort of situation, the rule that performs the attachment should also explicitly set the `insert` attribute appropriately, to override the default behavior:

```
gC {insert = false}  gB {attach {to = @1; ... }; insert = true};
```

By default, attaching a glyph moves the cursor to be following the attached glyph, otherwise the advance width of the base character is taken. Adjustment will also move the cursor but never so that the advanced width of the new position is negative, and never off the base line. This latter principle works well for diacritic adjustment since it is never necessary for the cursor to be moved backwards from the advance width of the previous character. In the unknown situations where this is required, direct kerning can be used.

4.6.6 Metrics

So far, no discussion has been made of how positional information is expressed. Numeric values can be scaled to the size of the font’s em square, or unscaled. Scaled numbers are specified by postfixing an ‘m’. By default the scaling factor is 1000, thus `500m` indicates 50% of em. In general, values related to glyph metrics should be scaled; the compiler will give a warning otherwise. Floating-point numbers are not allowed.

The scaling factor can be specified with the `MUnits` directive. The most common reason for changing the scaling factor would be to match the units per em square for the font a particular GDL was designed for. Such a scaling factor could make it easier to specify attachment points and ligature component boxes.

Metrics are available for all glyphs in a rule's context (using dot notation with a slot reference if needed) and when specifying glyph attributes for a class. The following metrics are available:

- `leftsidebearing` (`lsb`), `rightsidebearing` (`rsb`)
- `advancewidth` (`aw`), `advanceheight` (`ah`)
- `bb.left`, `bb.right`, `bb.top`, `bb.bottom` (`bb` is an abbreviation for `boundingbox`)
- `boundingbox.height` (`bb.ht`), `boundingbox.width`
- `ascent`, `descent` (as defined in the font)

The coordinate system for specifying attachment points, moving glyphs, etc. always increases left to right and bottom to top (i.e. a typical left-to-right system). The origin (0,0) corresponds to the left-hand side of the glyph on the baseline.

4.6.7 Substitution in a positioning pass

As of version 1.3.9 of the Graphite engine, it is permitted to include substitution rules within the positioning table. This can be useful if the shape or size of the substituted glyph is based on the position that has been determined within the positioning process.

For instance, in Nastaliq you might substitute shorter versions of kaf and gafs depending on their vertical position. Another idea would be to substitute smaller versions of diacritics based on the results of a collision avoidance pass.

Substitution rules within a positioning pass are more limited than those in the substitution passed. You can only do a one-to-one substitution—no insertion, deletion, or reordering. Also note that if an attachment existed involving the previous glyph, the new glyph will be considered to have the same attachments but the positions might now be wrong. None of the positions will be adjusted based on the new glyph.

4.6.8 Examples

The following are some examples of positioning.

4.6.8.1 Example 1: Lam-Alef

The first example is from Arabic and addresses the problem of ligatures with component diacritics. The ligature is Lam-Alef and there are diacritics which may need to go on the various components of the ligature. The two rules might be:

```
table(sub);
   gLam clsM1? gAlef clsM2?  >
      gLaf:(1 3) {component {lam.ref = @1; alef.ref = @3}}
      clsLM1 _ clsAM2;
endtable;

table(pos);
   clsLM1? {attach {to = @1; at = ldia; with = base}}
      clsAM2? {attach {to = @1; at = adia; with = base}}
         / gLaf _ _ ;
endtable;
```

4.6.8.2 Example 2: Dotless i with Tilde

The second example is taken from a Roman based font, such as IPA, in which there is a dotless i with a tilde over it. The tilde is wider than the i and for this example consider its advance width to be zero. Advance width modification is required for following letters with a high initial stem.

```
clsUDia {attach.to = @1; attach.at = udia; attach.with = base}
            / clsbase _ ;
clsWideDia {advance.x += bb.width/2} / clsIbase _ clsLeftStem;
```

Except that this does not deal with wide diacritics which may be placed under the i rather than on the top.

4.7 Placement

The final phase of processing is a cleanup and resolving operation, which occurs entirely within the renderer and in which the description file takes no part. This phase includes:

- Converting pseudo-glyphs to real glyphs
- Resolving positioning information, such as attachments, to absolute positions.

5 Example File

The following example fragment would be part of an IPA rendering description file. The fragment is concerned with two independent areas of rendering. The first is the handling of dotless i and friends. The second is the question of whether a user wishes to see pitch rendered as pitch letters or using superscript numbers. The alternative renderings are handled via a feature.

5.1 Example

```
/*
   Sample description for handling dotless i and raised numbers.
   Neither implementation is complete.
*/

#define c(x)    codepoint(x, 32765)//make a default codepage for IPA93
#define C(x)    codepoint(x)
#define u(x)    unicode(x)
#define p(x)    postscript(x)

#define LG_USENG 0x0409

table (glyph);

   // lists for dotless i substitution
   gOverTilde = c(226);
   clsTone = (c(157), c(152), c(147), c(143), c(136));
   clsUMod = (c(126), c(95), c(161), gOverTilde);
   clsUDia = (clsTone, clsUMod);
   clsDottedI =   (c("i"), c("j"), c(246));
   clsDotlessI =  (c(34),  c(190), c(174));

   // these lists are shortened for the example
   clsLStem = (c("DHLT[\\]bfghikl") c(132));
   clsTakesDia = (c(65 .. 71), c("I"), c(75 .. 86), c(88 .. 90),
            c(97 .. 123));
   clsIBase = (clsDottedI, clsDotlessI, c("l"));

   // lists for converting pitch letters to superscript numbers
   cls1Pitch = (c(159), c(154), c(149), c(145), c(138));
   cls2Pitch = (c(232), c(217), c(216), c(134), c(133), c(128));
   cls1Num = C("12345");      /* use real numbers */
   cls2Num1 = C("133551");
   cls2Num2 = C("515133");

   gRaise = pseudo(codepoint("^")); // make spare glyph to mark raises
   clsRaise = (u(0x0030 .. 0x0039), C("-"), c("nhjNm") c(248));
   clsRaised = (u(0x2070), u(0x00B9), u(0x00B2), u(0x00B3),
         u(0x2074 .. 0x2079), u(0x207B), u(0x207F),
         u(0x02B0), u(0x02B2), p("engsuperior"),
         p("msuperior"));

   // define attachment points
   clsDottedI {udia = point(advancewidth/2, bb.top + bb.top/5)};
   clsUDia {base = point(aw/2, 0);
         udia = point(aw/2, bb.top + bb.top/5)};
```

```
endtable; // glyph

table (feature);
   fPitchNum.id = 64000; // arbitrarily chosen id from user-dfnd range
   fPitchNum.name.LG_USENG = string("Pitch Numbers");
   fPitchNum.default = letters;
   fPitchNum.settings.letters.value = 0;
   fPitchNum.settings.letters.name.LG_USENG = string("Letters");
   fPitchNum.settings.numbers.value = 1;
   fPitchNum.settings.numbers.name.LG_USENG = string("Numbers");
endtable; //feature

table(sub);

pass(1);
   clsDottedI > clsDotlessI / _ clsUDia;  /* dotless i substitute */
   clsTone clsUMod > @2 @1;               /* diacritic then tone */
   if (fPitchNum == numbers)
      cls1Pitch _ > cls1Num gRaise;       /* just one raised number */
      /* others result in x3-5 or whatever - lots of insertion! */
      cls2Pitch _ _ _ _ _
        > cls2Num1 gRaise C("-") gRaise cls2Num2$1 gRaise;
   endif;
endpass; //pass 1

pass(2);
   clsRaise gRaise > clsRaised _ ;
endpass; //pass 2

endtable; //sub

table(pos);
   clsUDia {attach {to = @1; at = udia; with = base}}
      / clsTakesDia _;
endtable; //pos
```

5.2 Description

5.2.1 Macros

The example starts by using the C pre-processor to effectively allow us to work in two different encodings at the same time. The lowercase `c()` macro returns a glyph ID based on the IPA93 encoding. This assumes that the IPA93 encoding is a mapping to the correct Unicode values for those letters, rather than to some codepage 1252 overloading. The uppercase `C()` macro uses the default codepage (1252) to map standard ASCII type letters which are not available in IPA93.

5.2.2 Glyph Table

5.2.2.1 Glyphs and Classes

The next step is to define some variables. The first block of assignments are the classes needed for the dotless i substitutions. The next block contains three class assignments. These assignments would be very much longer in a real description file, but have been truncated in order not to swamp the example. The final block contains assignments to handle the change of pitch letters to superscript numbers.

The `pseudo` glyph has been created to help mark characters which should be superscripted. There is probably a better way of doing this (like converting to the superscript glyphs directly), but this illustrates a useful technique.

5.2.2.2 Glyph Attributes

Following these assignments, we have the glyph attribute assignments to indicate where the diacritic attachment points are on each glyph. Since the font has not been modified to add specific attachment points, locations are used instead.

5.2.3 Feature Table

Next we have a feature definition for the pitch numbers question.

5.2.4 Substitution table

After all this preamble, we are ready to write some rules. The substitution table is taken in two passes. The first pass does most of the work, dealing with dotless i, diacritic re-ordering, and then the possible conversion of pitch letters into numbers.

There are two sorts of pitch letters we need to consider. The first is a level pitch, which just gets converted to a single superscript number. The second is a simple contour between two pitches, which must be represented by a sequence of numbers: for example, [a^{3-5}]. This means that a single pitch letter glyph is converted into 3 output glyphs. We also use our pseudo-glyph to mark the numbers that need raising, which results in there being 5 glyphs inserted into the stream and 1 substituted.

One question which immediately leaps to mind when looking at these rules, is why there are no cursor associations for the inserted and deleted glyphs. Since in each case there is only one non-deleted or inserted glyph, there can only be one association possible. Therefore, there is no need to explicitly give a long list of associations which are clearly obvious. Even if the fallback effects of not associating, as given in the next section, come into play, the results will still be obvious to the user.

The second pass simply deals with the special raising glyph. Anything followed by this glyph is converted to a raised form. The result is that there is no possibility for the raised glyph to appear in the final output. Our initial mapping of the glyph to a ^ glyph, hopefully, will never come into play.

5.2.5 Positioning Table

The positioning table attaches a diacritic to its base character. In fact, the base character could be another diacritic, which is why diacritics need two attachment points: one to attach with and one to have others attach to.

5.3 Conclusion

This GDL file does a lot of work in a remarkably small number of rules. In a real situation, especially for something as complex and quirky as rendering IPA, there would be many more rules and many more classes. It behooves the GDL author to use standard programming techniques to organise their description. Again, the C pre-processor can help here by allowing some of the information to be stored in external files which are #included into the main description.

6 Advanced Concepts

The rest of this description examines more detailed and advanced aspects of the Graphite description format. It looks at the description from both a more computer scientific standpoint and from its implementation.

6.1 Cursor Hitting

The association model which is presented as part of the substitution rules is not strictly correct. From a descriptive standpoint, association is a helpful way of considering what is going on. But from a cursor point of view, it is not the glyphs themselves which are in focus, but the cursor points between the glyphs.

6.1.1 Split Cursors

When considering a cursor between two characters in a stream of text, we can say that it is after one character and before the next both on the surface and in the underlying data. The problem is that when the underlying to surface relationship becomes more complex, a cursor may not be between adjacent glyphs. Consider an internal cursor placed between two codes in an underlying string. On the surface, the two glyphs that the underlying codes are associated with may not be adjacent. The result is that on the surface, it is necessary to split the cursor to indicate which glyphs the cursor is before and after.

As an example, consider the word *tirkha*, meaning 'thirst' in Nepali. In its underlying form, it is stored as **tirkha**, but on the surface, it is rendered with the **i** before the **t** and the **r** placed above the ***a*** which is placed after the **kh** (a single glyph – aspirated k). The result is **itkhar**:

If in the underlying form we were to place a cursor between the **t** and the **i** we would get a split cursor on the surface to indicate that we were after the **t** and before the **i**. This would look something like:

The before cursor is placed low down and the after cursor high up, following the German convention with quotation marks. The serif on the cursor helps to show which glyph it is relevant to. Worse, if the cursor were placed between the **i** and the **r**, we would get:

Here the before cursor is before the clump containing the **r**.

6.1.2 Basic Principles

One of the basic principles of split cursors is that *you can only be at one place in the underlying text*. Here is a typical approach to cursor placement in a complex rendering system.

- The user clicks the cursor down somewhere on the text on the screen.
- The application asks Graphite where the cursor has been placed.
- Graphite calculates a position in the surface glyph string for the cursor, including in this attached glyphs, etc. It then returns either an underlying position the cursor is before or an underlying position the cursor is after depending on where the click occurred.
- Graphite returns the position for the split cursor.
- The application tell Graphite to show a split cursor on the screen.

Another important principle is that *editing is done on the underlying text*. This means that handling backspace is the duty of the keyboard handler in conjunction with the underlying text directly. The renderer only deals with rendering that edited underlying text. The keyboard handler does not work via the renderer since the rendered form of the text is not held anywhere, except for display purposes.

Arrow keys should endeavor to move through the surface text, although to what extent this relates to attached glyphs is up to the application. Arrow keys should also endeavor to move in the direction indicated regardless of directionality of the text. This requires interaction with Graphite.

6.1.3 Before & After

After Graphite has finished dealing with all the associations, insertions, deletions, etc. the final result is for Graphite to know, for any position between two characters in the underlying text, what positions in the surface text this underlying position is before and after. Likewise it also knows the reverse information of how a position between two surface glyphs maps to before and after positions in the underlying text.

6.1.3.1 Insertion & Deletion

The default behavior of inserted and deleted glyphs which have not been associated is not immediately obvious. As a result, it is best never to rely too much on the default behavior. By looking at a rule it is almost impossible to work out all the implications of the defaults without running the cursor tracking algorithm by hand. Having said this, the results of the default behavior are very natural and may be relied on to give some sort of behavior which a user might expect.

For completeness, though, the default behavior is described here.

6.1.3.1.1 *Insertion*

An inserted glyph is not accessible from the underlying text. There is no underlying cursor position which maps to a position which can interact with the inserted glyph.

On the surface, placing a cursor before an inserted glyph results in an underlying position before the following glyph. Likewise placing the cursor after an inserted glyph results in an underlying position after the previous glyph to the insertion point in the underlying text.

6.1.3.1.2 *Deletion*

A deleted glyph is not accessible from the surface text. There is no surface position which results in an underlying position before or after the deleted glyph.

In the underlying text, a position before the deleted glyph results in a surface position of before the following glyph, and a position after the underlying glyph results in a surface position after the previous glyph.

6.1.4 Insert Attribute

The `insert` slot attribute impacts how cursor tracking works. By default all slots have this attribute set to true. When attachment is done, `insert` is set to false. Of course, `insert` can also be set explicitly to 1 (true) or 0 (false).

When `insert` is false on a slot, the cursor is never placed between the (first) corresponding character and the character that preceeds it. Note that the `insert` attribute really affects the corresponding *character,* since that is what insertion bars are associated with, not the glyph itself.

When the user clicks at a location where `insert = false`, Graphite will move the insertion to one side or the other, to the closest legal insertion point. Similarly, if the application program tries to set an insertion point at a place in the underlying text that would correspond to one of these invalid locations, Graphite will suggest an alternate legal position. (However, ultimately it is up to the application whether or not it abides by the insertion information Graphite provides.)

6.2 Pseudo-Glyphs

The `pseudo` directive synthesizes a new glyph, just as if the font designer had copied an existing glyph to an unused slot in the font and assigned it a Unicode value.

Consider an example: suppose a script-engineer wants to support a special variant of A which has been given a PUA allocation of `0xf141`. He could be tempted to map it straight to the glyph `u(0x0041)`. But, apart from not being able to, he wants to be able to position this new glyph differently from `u(0x0041)`. So he includes the following statement:

```
GP1 = pseudo(unicode(0x0041), 0xf141)
```

This command does two things. First, it has the effect of creating a new glyph in the font. It finds a spare glyph number (assuming the font hasn't filled its 64K allocation of glyphs, in which case an error is raised) and assigns this to `GP1`. Second, when initially processing the input, it maps the Unicode codepoint `0xf141` to the pseudo-glyph. At the very end of processing, it will convert any instances of this pseudo-glyph to be the glyph associated with U+0041. In other words, our pseudo-glyph will look like an A.

In fact, if two Unicode codepoints are mapped to the same glyph by the `cmap`, one of them will be automatically mapped to a pseudo-glyph and then mapped back at the very end. This ensures that two codepoints can be treated differently within the rule matching which follows. This auto generation of pseudo-glyphs can be disabled by assigning `0` to the `AutoPseudo` setting at the beginning of the GDL file.

Notice that the `unicode` and `codepoint` functions will return the glyph that the Unicode value has been mapped to within the program. This may be a pseudo-glyph ID or a real glyph ID. For the most part this is the expected behavior. But should a GDL author require access to the real glyph ID of a glyph, regardless of whether it is pseudo or real, he can use the `glyphid` function which guarantees to return the real glyph ID (the one which a pseudo glyph will revert to at the end of all the processing).

As we saw in the example, the `pseudo` function does two things: creates a pseudo-glyph mapped to a real glyph during output, and maps a Unicode codepoint to a pseudo-glyph. It is not always necessary to do the latter mapping, and `pseudo` may be used with just one parameter to create a pseudo-glyph mapped to a real glyph.

```
pseudoX = pseudo(unicode(0x002C))
```

This is useful if two identical glyphs need to be rendered with the same glyph but positioned differently. The positioning rules may not be able to express the complex contexts involved and using a different glyph may fix the problem.

The following statements:

```
p1 = pseudo(u(0x002C), 0x201A)
p2 = pseudo(u(0x00AE), 0x201A)
```

are in error. It is an error to try to manually map a Unicode codepoint twice. The automatic creation of pseudo-glyphs can be overridden but not twice.

6.3 User-definable Slot Attributes

In addition to the slot attributes mentioned above, there is also a set of user-definable slot attributes that can be used in any way the programmer deems helpful. The names of these slot attributes are `user1`, `user2`, `user3`, … `user64`. It is generally most helpful to use the `#define` mechanism to give the attributes more meaningful names.

These slot attributes can be used to communicate information between passes. For instance, one pass might set a flag based on the sequence of glyphs it encounters, and a subsequent pass could perform a substitution or adjust the position of glyphs based on the value of the flag.

```
#define raiseFlag user1


table(sub)
   // record the fact that the tone mark needs to be raised,
   // and delete the character that should not be displayed
   clsToneMark gRaiseMark  >  @1 { raiseFlag = true } _;
endtable;

table(pos)
   // shift the tone mark up if the raised flag is set
   clsToneMark { shift.y = 100m } / _ { raiseFlag == true };
endtable;
```

Note: in order to minimize the amount of memory required by the Graphite engine, it is strongly recommended that you use consecutive, low-numbered user-definable attributes rather than an arbitrary set of these. For instance, it is preferable to use the following:

```
#define vowelMarker user1
#define diacMarker user2
#define consMarker user3
```

as opposed to:

```
#define vowelMarker user10
#define diacMarker user56
#define consMarker user28
```

6.4 Backing up the Stream Position

Due to the way Graphite manages the process of matching rules, it is possible to write rules that cause the stream position to move backwards. Consider the following:

```
clsVowel > clsVowelAlt / ^ clsCons _;
```

Before this rule is matched, the position of the stream is considered to be just before the vowel, but after the rule fires, the position of the stream is before the preceding consonant.

In order to allow this phenomenon to occur, it is necessary to set the `MaxBackup` directive to some positive number. `MaxBackup` should be set to the number of successive slots that need to be backed over as a unit. If `MaxBackup` is not set high enough to handle a sequence of back-up operations, the processing will simply keep the position of the stream unchanged. For instance, if the above rule is included in a pass and `MaxBackup` equals zero, the stream position will not be set before the consonant, but will be left before the vowel. You will likely need to set `MaxRuleLoop` to at least twice the value of `MaxBackup`.

6.4.1 Example

This back-up mechanism can be used to handle a sequence of modifications that are based on first recognizing the end of the sequence. Suppose you want to change a sequence of the letter A to alternate between two forms, A1 and A2, but with the final item always being A2 regardless of whether there is an odd or even number in the sequence. You can use the approach of first recognizing the end of the sequence and using the back-up mechanism to modify each previous item based on the following one. `MaxBackup` should be set to the maximum expected length of the sequence.

```
// For marking the elements of the chain of alternating items:
// 0 = not in chain; 1 = change to A1; 2 = change to A2
#define Alt user1

table(sub) {MaxRuleLoop = 20; MaxBackup = 10}

   // Beginning of sequence: another A follows this one;
   // keep going forward till we hit the end of the sequence:
   gA  >  @   /  _  gA {Alt == 0};

   // Found the end of the sequence; start a chain and back up;
   // mark this first A to be changed to A2:
   gA   >  @ {Alt = 2} / ^ ANY  _ {Alt == 0};

   // Continue backwards; mark this A the opposite of the
   // following one:
   gA  >  @ {Alt = 2}  /  ^ ANY  _ {Alt == 0}  gA {Alt == 1};
   gA  >  @ {Alt = 1}  /  ^ ANY  _ {Alt == 0}  gA {Alt == 2};

   // Special case: hit the beginning of the sequence with
   // no glyph before:
   gA  >  @ {Alt = 2}  /  ^ _ {Alt == 0}  gA {Alt == 1};
   gA  >  @ {Alt = 1}  /  ^ _ {Alt == 0}  gA {Alt == 2};

   // When going forwards: switch to the alternate form:
   gA  >  gA1  /  _ {Alt == 1};
   gA  >  gA2  /  _ {Alt == 2};

endtable;
```

6.5 Justification

Graphite includes various mechanisms to allow a range of text to be fully justified to a specific width as requested by the application. Justification can be performed by techniques such as kerning to adjust the amount of space between glyphs, insertion of kashidas (extender glyphs to create stretch within cursive script), substitution of glyphs of varying widths, and creation or removal of optional ligatures.

Justification may involve either stretching or shrinking the line to fit a given amount of space. The techniques to stretch and shrink a given glyph may be quite different. For instance, it may be possible to stretch a glyph by following it with kashidas, but shrinking is not possible using this technique, and must be achieved through kerning or glyph substitution, or may not be possible at all.

Keep in mind that many applications that support full justification use only stretching, not shrinking. In other words, they never attempt to fill the line beyond what will naturally fit, so shrinking is never necessary. For this reason, shrinking is most useful within applications providing high-end, sophisticated paragraph and text layout.

6.5.1 Justification Overview

Justification is performed by setting justification-related glyph attributes and adding appropriate rules to the rule tables. These attributes and rules take effect in two stages. In the first stage, each glyph is assigned a potential stretch or shrink, indicating how much it is possible to adjust the width of the glyph. The second stage involves using the actual assigned width to modify the glyphs and actually achieve the desired width.

More specifically, justification is incorporated into the Graphite processing model as follows:

- Stage 1
  - Glyphs are initialized with *glyph attributes*, which may include those related to justification.
  - The *substitution table* may include rules to set the stretch and shrink values of each glyph and related information. Note that this is the *potential*, or maximal, adjustment, not the exact adjustment.
- The bidi table is run as normal.
- After the bidi table, the *justification routine* determines where adjustments should occur to create the necessary width. This routine is implemented by the application, and therefore may differ somewhat in its exact effects from one application to another.
- Stage 2
  - The *justification table* is a special substitution table that runs following the justification routine. Its purpose is to perform substitutions that are needed to achieve justification, such as replacing a narrow glyph with a wide glyph, or inserting kashidas.
  - The *positioning table* may include rules to adjust the positions of glyphs as needed for justification.

6.5.2 Default Basic Justification

The Graphite system provides basic white-space justification, involving the capacity to stretch white space up to 100 times its natural width and shrink it to 75%. This behavior is implemented at level 0, the "emergency stretch" level (see the discussion of justification levels below). You may override this behavior in your GDL program if you so desire.

6.5.3 Global State Variables

Global state variables are available to test the state of justification-related processing. These variables can be used within rule constraints to determine which justification-related rules, if any, should be fired.

6.5.3.1 JustifyMode

The JustifyMode variable indicates the justification mode in which the engine is being run. In other words, it indicates whether and how the application is interacting with the Graphite engine in order to generate justified text. There are three possible modes:

- `JMODE_NORMAL` – no justification is desired; the justification routine is not run.
- `JMODE_JUSTIFY` – the justification routine will be run in order to produce justified text.
- `JMODE_MEASURE` – used by applications that are doing sophisticated high-end justification. The "measure" mode allows the application to measure the width of text before actual layout in order to determine where to place line-breaks.

6.5.3.2 JustifyLevel

Eventually we anticipate four possible levels of justification rules, depending on how much adjustment is needed, and JustifyLevel indicates which level is being applied. It is generally considered during Stage 2 to determine which rules to fire. The standard levels are 1, 2, and 3, where in general the higher level represents the more extreme or invasive approach. Level 0 represents an "emergency level", and as such the application may decide to take a different approach than what is specified by the GDL rules. See the discussion of justification levels below.

Note: as of Graphite2 version 1.3.12, only one justification level is supported. The JustifyLevel variable should not be used.

6.5.4 Stage 1: Specifying Potential Stretch and Shrink

There are several attributes that can be used to indicate how much, and in what ways, a glyph is permitted to stretch and shrink. These exist as both glyph and slot attributes. The `justify.stretch` and `justify.shrink` attributes indicate the maximum amount by which the glyph can be stretched and shrunk, respectively. The value is in em units. For example, the following indicates that a space character can be stretched to 10 times its natural width (increased by 900%) and shrunk to 75%.

```
gSpace {justify {stretch = aw * 9000; shrink = aw / 4}};
```

Note that the values of these attributes indicate the amount by which the width can be *adjusted,* not the total final width. Also note that the value of `justify.shrink` is always positive.

In some cases width can be adjusted only in increments. For instance, when inserting kashidas, the adjustment must be made in strict multiples of the width of the kashida. The `justify.step` attribute can be used to indicate this; its value is the width of the increments. For example, the following permits the insertion of up to 5 kashidas:

```
gKashida { incWidth = aw };

clsLetter {justify { stretch = gKashida.incWidth * 5;
                     step = gKashida.incWidth }};
```

When substituting one glyph for another, `justify.step` can be used to indicate that the adjustment must be exactly the difference of the widths of the two glyphs:

```
gNarrow { xWid = aw };

gWide { justify { shrink = aw - gNarrow.xWid;
                  step = -justify.shrink}};
```

If the step value is positive it applies when stretching, and if negative, it applies when shrinking. If a step value is needed for both stretching and shrinking, two separate justification levels must be used (although this is not supported as of Graphite2 version 1.3.12).

The `justify.weight` attribute can be used to indicate that some glyphs should be given preference in deciding how to distribute width adjustments. Assigning a glyph a weight of 10 means that it will receive 10 times as much adjusted width (if possible, given its total stretchability) as a glyph with weight 1. The default weight is 1, and the maximum weight is 255. In the following example, we use the `justify.weight` attribute to prefer stretching of white space over intra-word stretch:

```
gSpace { justify { stretch = aw * 9000; weight = 10 }}

clsWordForming { justify.stretch = 100m } // default weight = 1
```

There are slot attribute equivalents for each of these attributes that can be used within rules. For instance, the following rule uses kerning in sequences such as "WA" and "VA" to remove the illusion of white space between the diagonal strokes.

```
(gW gV) { justify.shrink = 100m }  /  _  gA;
```

6.5.4.1 Trailing White Space

If part of your strategy is to stretch white space, you will need to include a rule to remove the stretch from white space occurring at the end of the line, since it is not part of the line's visible width. The following is an example of such a rule that will handle up to five trailing space characters:

```
gSpace {justify.stretch = 0}
   / _  [gSpace [gSpace [gSpace gSpace? ]? ]? ]?  #;
```

(Note that this happens automatically for the built-in white-space-stretching capability that is provided at level 0.)

6.5.5 Stage 2: Performing Justification

The justification routine, which is run just after the bidi pass, sets the `justify.width` attribute for each stretchable or shrinkable glyph to the desired amount of adjustment. This value is used during stage two—within the justification and positioning tables—to determine how, and how much, to modify the glyphs.

Rules to handle justification by kerning are placed in the positioning table, along with all the other positioning rules. A simple example is shown below. It is good practice to test the JustifyMode variable to ensure that rule is only fired when justification is needed:

```
table(pos)

if (JustifyMode == JMODE_JUSTIFY)
   someGlyph {adv.x += justify.width; justify.width = 0};
endif;

endtable;
```

It also is good practice to subtract from `justify.width` any width that is being handled by the rule, so here we set the value to zero. This is not actually necessary when your program includes nothing but simple justification, but it becomes more important when you begin working with multiple rules and strategies.

Justification-related substitutions, insertions and deletions are performed in the justification (or "just") table, which is run after the substitution table and bidi pass (if any) and before the positioning table. Again, you will want to test the value of JustifyMode so that the rule is fired only when justification is necessary.

```
table(just)

if (JustifyMode == JMODE_JUSTIFY)
   gStandard  >  gWide
      / _ {justify.width >= justify.stretch};
endif;

endtable;
```

The rule above substitutes a wide version of a certain character for the standard version. The rule uses the constraint to make sure that the amount of additional width assigned to the original glyph is at least equal to the amount of stretch that will be achieved by making the substitution.

A more complete example below shows how to adjust the width of a glyph using a combination of substitution and positioning. The amount of stretch available equals the difference between the wide and standard glyphs, plus a small amount of additional kerning.

```
table(glyph)

   gStandard {
      wideGlyphDiff = gWide.aw - aw;
      justify.stretch = wideGlyphDiff + 100m};
```

```
endtable;

// no rules needed in the substitution table

table(just)

   // rule 1
   gStandard  >  gWide {justify.width -= wideGlyphDiff}
      /  _ {justify.width >= wideGlyphDiff};

endtable;

table(pos)

   // rule 2
   (gStandard gWide) {adv.x += justify.width; justify.width = 0};

endtable;
```

This example shows why it is a good practice for each rule to subtract the "handled" width from `justify.width`. Rule 1 subtracts the width handled by virtual of substituting the wide glyph, leaving the remaining width to be handled within the positioning table using kerning (rule 2).

Notice that (because `justify.step` is not set), the assigned width may be less than `wideGlyphDiff`, in which case all the stretch will be handled by rule 2.

6.5.6 Tips and Tricks

6.5.6.1 Kashida Insertion

To use kashida insertion to accomplish justification, you would set the `justify.step` attribute to the width of the kashida that can be inserted. The `justify.stretch` attribute will generally be set to a multiple of the width of the kashida, the number of kashidas that can be inserted. These attributes could be specified either in the glyph table alone or also using a rule in the substitution table. Then the justification table will contain the rules to actually insert the kashida, and the number of kashidas to insert would be based on the value of the `justify.width` attribute (as it was set by the justification module which happens between the running of those two tables).

The following shows an example where a feature is used to control the amount of stretch permitted. Note that the rule in the justification table uses the scan position adjustment mechanism to repeatedly insert kashidas until all the assigned width has been accounted for. Setting MaxRuleLoop to something relatively high is useful when this mechanism is operational.

```
table(glyph)

   gKashida = glyphid(…) { xAdv = advancewidth };

   clsCanTakeKashida { kStretch = gKashida.xAdv;
                       justify.step = kStretch };

endtable;

table(subs)

// The 'stretch' feature indicates how much stretch we allow.
if (stretch == maximum)
   clsCanTakeKashida { justify.stretch = kStretch * 5 };

endif;
if (stretch == medium)
   clsCanTakeKashida { justify.stretch = kStretch * 3 };
```

```
endif;
if (stretch == minimum)
   clsCanTakeKashida { justify.stretch = kStretch };
endif;
// if (stretch == none), leave justify.stretch = 0.

endtable;

table(just) {MaxRuleLoop = 30}

   // Keep inserting as many kashidas as there is width for:
   clsCanTakeKashida  _
      >  @1 {justify.width -= kStretch}  gKashida:1
      /  ^ _ {justify.width >= kStretch}  _;

endtable;
```

6.5.6.2 Ligature Expansion

The `justify.step` attribute is useful to perform ligature expansion as well. In this case the width of the step is exactly equal to the difference between the ligated and non-ligated forms. In a normal mode of operation, the substitution table is used to create the ligatures, but when justification is occuring, it is the justification table that must be used, so it can recognize when *not* to create the ligatures. Unfortunately, this leads to a slight duplication of code.

```
table(glyph)

   // ligatures
   g_ae { xAdv = advancewidth; kStretch = 190m };
   g_oe { xAdv = advancewidth; kStretch = 210m };

   g_e { xAdv = advancewidth };

   g_a { ligDiff = advancewidth + g_e.xAdv - g_ae.xAdv;
         kStretch = 150m };
   g_o { ligDiff = advancewidth + g_e.xAdv - g_oe.xAdv;
         kStretch = 200m };

   g_ae { ligDiff = g_a.ligDiff };
   g_oe { ligDiff = g_o.ligDiff };

   clsMakesLigWithE = (g_a, g_o);
   clsELig = (g_ae, g_oe); { /* define component boxes */ }

endtable;

table(subs)

if (JustifyMode = JMODE_NORMAL)

   // Normal case: always make the ligature.
   clsMakesLigWithE  g_e
      >  clsELig:(1 2) { /* define component refs */ } _;

endif;

if (JustifyMode = JMODE_JUSTIFY)

   // This stretch value assigned here is valid when we *could* create
   // a ligature.
   clsMakesLigWithE  { justify { stretch = ligDiff + kStretch[1];
                                 step = ligDiff }
      g_e;
```

[1] Notice a slight anomaly here. We assign extra kerning width `kStretch` assuming that the ligature will not be created. If in fact the ligature is created, and `kStretch` is substantially greater than the value for the ligature, the ligature may

```
    // Otherwise assign the normal kerning stretch value.
    clsMakesLigWithE  {justify.stretch = kStretch };

endif;

endtable;

table(just)

if (JustifyMode = JMODE_JUSTIFY)

    // Only create the ligature when we DON'T want to stretch.
    // Note that in this case, the step mechanism should ensure
    // that justify.width = 0.
    clsMakesLigWithE  g_e
       >  clsELig:(1 2) { /* define component refs */ }  _
       /  _ {justify.width < justify.stretch}  _;

    // Subtract the amount of stretch we "inserted" by virtue of having
    // NOT created the ligature. The extra width will be handled by the
    // positioning pass. (Note that due to the step mechanism the
    // extra width will be exactly ligDiff, or zero.)
    clsMakesLigWithE  { justify.width -= ligDiff }  g_e
       /  _ { justify.width >= justify.stretch }  _;

endif;

endtable;

table(pos)

if (JustifyMode = JMODE_JUSTIFY)

    clsMakesLigWithE  { kern.x = justify.width };

endif;
```

To avoid the duplication of the rule, it would be possible to create the ligature and undo it later. This would result in a loss of the correspondences between the non-ligated glyph forms and their underlying characters.

As the comments above mention, the step mechanism will constrain the kerning width that actually gets assigned to be a multiple of the `justify.step` value. The multi-level justification capability that has not yet been implemented would provide an improvement.

Note that together all the `justify.width` values set by the justification module should produce cleanly justified text. It is essential that the subsequent justification and positioning passes be meticulous about making adjustments to account for each value. Failure to do so will result in improperly justified text.

6.6 Mirroring

In right-to-left scripts, certain characters need to be displayed as mirrored alternates. For instance, while in a left-to-right script the opening parenthesis is displayed as "(", in a right-to-left script it should appear as ")". The same is true for similar pairs of characters such as brackets ("[…]"), braces ("{…}"), and wedges ("<…>").

---

be kerned inappropriately. This would most likely be a problem when the step value is relatively small and the difference between the two `kStretch` values is rather large. A multi-level justification approach would be needed to solve this problem cleanly.

The `mirror.glyph`glyph attribute can be used to specify what form a mirrored glyph should take. The value of the attribute is the glyph number of the alternate glyph. In the Graphite2 engine, the bidi pass will use these attributes to perform mirroring.

Note: the mirror attributes are only handled by the Graphite2 engine. There are no plans to support automatic mirroring in the original Graphite engine.

For the examples mentioned above, the alternate glyph shapes come in pairs that can represented by pairs of Unicode characters (e.g., U+0028/U+0029, U+005B/U+005D, etc.). The mirroring can be accomplished by substituting the glyph normally assigned to the opposite member of the pair. The Unicode Standard defines these pairs, and the Graphite compiler will set the `mirror.glyph` attributes automatically based on the information in Unicode. These values can also be overridden in GDL code as necessary.

In other cases, a single glyph exists that needs to change its shape in a right-to-left context. There are quite a few examples of these among mathematical symbols, such as the square root sign. There is no Unicode character that represents the alternate form of the square root; it is simply an alternate shape of the symbol. In these cases the `mirror.glyph` attribute must be set in the GDL explicitly, since there is no way of determining a default value from Unicode.

6.6.1 Application-based mirroring

Some applications perform mirroring independent of Graphite. In this case the Graphite engine does not want to duplicate the work that was already performed by the application. However, the application is only capable of mirroring encoded pairs such as parentheses and brackets, not the single mirrored characters such the square root symbol.

The `mirror.isEncoded` glyph attribute exists to indicate which glyphs should always be mirrored versus those that should only be mirrored when they have not already been handled by the application. Appropriate values are 0 (false) and 1 (true). Like `mirror.glyph`, this attribute is set automatically by the Graphite compiler, and can be overridden in the GDL code.

6.7 Pass optimizations

6.7.1 Pass constraints

Pass constraints provide a way to optimize your code so that some passes are ignored altogether. For instance, you may have a pass that only contains rules to handle a specific font feature. By embedding the pass inside a constraint, the pass can be skipped efficiently.

A constraint is considered to apply to a pass when the `if` statement containing the constraint is followed immediately by a `pass` statement.

```
if (specialFeature)
pass(3)

   // rules to handle specialFeature

endpass;
endif;
```

If the structure of the code inside the `if` statement is such that it contains something other than `pass` structures, the constraint will be applied directly to each of the rules rather than at the pass level. Note that the behavior will be the same, but the optimization effect is lost.

6.7.2 Pass key slots

The compiler implements an optimization that allows the engine to skip passes that are not needed. To do this, it creates the set of key glyphs by looking at each rule in the pass. For each rule it works out which glyphs, if not present, mean that the rule would never match. For example, if the pass consists of a rule to create an fi ligature from the characters f and i, any glyph other than an f or an i means that the rule will never match. More than that, we can say that if the run does not contain an f (regardless of whether it has an i), then the rule will not match. The compiler, therefore, analyzes the pass by looking at each rule, picking one slot and determining the set of glyphs that covered by that slot. If those glyphs exist in the run then the rule might match and the pass must be executed. If not, the pass can be skipped.

The mechanism used to identify the set of glyphs for this pass optimization is a built-in slot attribute called `passKeySlot`. Each rule in the pass will have one slot identified as the "key slot", and all of the glyphs that can possibly be in that slot when that rule fires are added to the set of glyphs used in the test to see if a pass can be skipped.

If the GDL author doesn't explicitly identify which is the key slot, the Graphite compiler will pick the first "modifiable" slot in the rule (i.e., the first slot in the left- or right-hand side of the rule, not in the context).

The default chosen by the compiler may not be optimal, however, so the GDL author can explicitly identify the key slot by setting the `passKeySlot` attribute to "true" (or 1).

6.7.2.1 Example

Consider a pass that attaches marks to their bases. For many scripts, the set of bases is much bigger and much more likely to occur in a run than the set of possible marks. By making the key slot be the slot containing the marks, we will get to skip the pass more often. A rule to do this might look like the following:

```
clsBase clsDia {attach {to=@1; at=udap; with=lap}; passKeySlot = 1};
```

Notice that the effect of setting `passKeySlot` is to indicate the second slot—the diacritic—as the key slot for the pass rather than the first, the base.

6.8 Automatic Collision Avoidance

As of version 1.3.0, the Graphite2 engine provides the ability to perform automatic collision avoidance. The GDL language includes directives to control the mechanism and attributes to parameterize the behavior.

Setting a pass's directive `CollisionFix` to something greater than zero causes the algorithm to be run at the end of the pass. The value of the directive indicates how many iterations the algorithm should use. A larger value may produce a better result in the case of complex collisions, but may also slow down the rendering process.

There are two aspects to the collision avoidance mechanism, shifting and kerning. A given pass can include just shifting or both shifting and kerning. It is possible to turn on collision avoidance for a pass that includes rules, or to use separate pass with no rules (the latter may be more convenient for debugging in Graide), in which case the collision fixing happens after the rules are run.

6.8.1 Shifting

The core of the automatic collision avoidance algorithm involves shifting "moveable" glyphs to avoid collision with base glyphs and other moveable glyphs. "Moveable" glyphs may include diacritics and other parts of characters that are represented by a separate glyph, such as nuqtas.

There are various factors that contribute to the glyph adjustment result, and each possible resulting position has a cost associated with it. The factors that are considered are as follows:

- The distance the glyph is moved from its original position—the algorithm attempts to minimize the total movement, using a weight of 1.
- How much the glyph intrudes into the specified margin of a neighboring glyph (`collision.marginweight`).
- How much the glyph violates the specified sequencing by intruding into undesirable spaces close to neighboring glyphs (`sequence.above.weight` and `sequence.below.weight`). See Sequencing, below.
- Whether the glyph and a neighboring glyph would create an undesirable horizontal line (`sequence.valign.weight`).

The attributes indicated above allow the GDL programmer to adjust the cost factors. These attributes can be set directly as slot attributes, or as glyph attributes from which slot attributes are then initialized.

There are a set of flags stored in `collision.flags` that specify how the glyph should be handled by the collision avoidance algorithm. Glyphs can be marked to be shifted (appropriate for nuqtas and diacritics), kerned (see Kerning, below), considered when fixing other glyphs but not moved themselves (appropriate for base glyphs), or completely ignored.

The flags are also used to indicate the start and end of a cluster of glyphs within which there is likely to be collisions to be fixed by shifting (rather than kerning). For instance, in Arabic script, it would be normal (although not required) to treat whitespace as the boundary of a cluster, so that collisions within the cluster are fixed using shifting and inter-cluster collisions are fixed using kerning. Clusters *can* be arbitrarily large, but keep in mind that using large clusters could potentially have a deterimental effect on the performance of the font, since the number of comparisons required is $O(n^2)$ where n is the size of the cluster.

The kerning process (described below) also takes into account a flag can mark certain glyphs as whitespace. Glyphs with no outline are automatically considered whitespace, but you might want to have "visible space" characters that are also treated as spaces. See the "collision.flags" attribute in the Reference section below.

Each glyph can be given an ideal margin using `collision.margin`. There is also a "movement rectangle" that indicates how far the glyph can be moved in any direction. This is controlled by `collision.max.x/y` and `collision.min.x/y`. Note that even for right-to-left scripts, positive numbers indicate right-ward movement and negative numbers indicate left-ward movement; this behavior is different from `shift.x` (where the direction is based on the direction of the script).

The Graphite compiler creates an estimate of the shape of the glyph for the purposes of collision avoidance. In the case of simple, convex glyphs, a simple estimate is usually adequate. But for glyphs with concave edges, a more complex estimate is often needed. Setting the `collision.complexFit` attribute to true creates the more complex estimate. (Tip: it can be surprising how subtly concave an edge can be to necessitate a complex estimate, in order to get high-quality positioning.)

It is also possible to adjust the glyph shape used for the collision algorithm using the `collision.exclude.glyph` attribute. The shape of the `exclude.glyph` is added to the shape of the main glyph when determining collisions or potential collisions.

6.8.1.1 Sequencing

The sequencing mechanism was designed specifically to help support the diagonal sloping effect that is distinctive of Nastaliq-style Arabic. It also can be used to enforce the position of diacritics relative to nuqtas.

The `sequence` attributes work by defining classes of glyphs and their behavior relative to glyphs in the same class or other classes. There are two kind of relationships.

6.8.1.1.1 *Vertical sequencing*

Certain kinds of glyphs can be kept above or below other glyphs. For instance, you might want to ensure that upper diacritics are be positioned above upper nuqtas, and lower diacritics below lower nuqtas—in spite of what might be permitted by their movement rectangles.

This is achieved by using `sequence.class` to create the two classes of glyphs, and setting `sequence.order` on the constrained glyph and its `sequence.proxClass` to the other class. The classes are identified by arbitrary integers. The built-in `order` constants are:

- NOABOVE = 4
- NOBELOW = 8

For instance, to ensure upper diacritics remain above upper nuqtas:

```
#define UNUQTA 1   // arbitrary
#define UDIAC 2    // arbitrary
#define NOBELOW 8  // built-in constant
c_upperNuqta {sequence.class = UNUQTA};
c_upperDiac {sequence {class = UDIAC; proxClass = UNUQTAS;
      order = NOBELOW}}
```

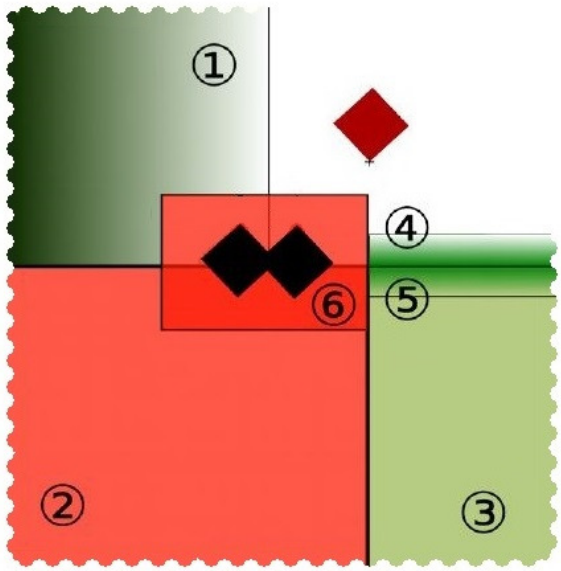
Note that it probably does not make sense to use the NOABOVE (or NOBELOW) flag *without* also setting `proxClass`. When `proxClass` is left unset, the `sequence.class` itself is used in its place. This would indicate that no member of the class can be above (or below) other members of the same class, which is inherently somewhat contradictory.

6.8.1.1.2 *Diagonal sequencing*

The sequencing mechanism was designed specifically to help support the diagonal sloping effect that is distinctive of Nastaliq-style Arabic by ensuring that a class of glyphs follow a diagonal sequencing pattern. The built-in `order` constant LEFTDOWN = 1 is used to generate a diagonal slope. (There is also a RIGHTUP constant, 2, but there is no known use for it, except internally within the Graphite engine.)

The `sequence` attributes are used to define regions around glyphs that are not appropriate for neighboring glyphs—as well as the cost of violations. Specifically, when you want to create a diagonal sequence of glyphs, the area below and on the far side of the glyph in question is not appropriate for preceding glyphs.

In the image below, we are trying to position the red single nuqta appropriately relative to the two black nuqat. The various regions show the costs of positioning the origin of the red nuqta, at the bottom corner shown by a very small +.



The various regions behave as follows:

Region 1. Do not position the target to the upper left of the neighboring glyph (the double nuqat). The further left you go the worse the result. The right side of Region 1, relative to the position of the neighboring glyph, is indicated by `sequence.above.xoffset`, and the cost is weighted by `sequence.above.weight`.

Region 2. Do not position the target to the lower left of the neighboring glyph. This is such a bad place to be that it is treated as a full exclusion, just as if the area were covered in ink from another glyph.

Region 3. Try not to position the target to the bottom right of the neighboring glyph. This isn't such a bad place to be, so the cost is flat in that it merely becomes more expensive the further the glyph is from its original position but with a weight multiplier. The left side of Region 3 (right side of Region 2) is indicated by `sequence.below.xlimit`, relative to the position of the double nuqta, and the cost is weighted by `sequence.below.weight`.

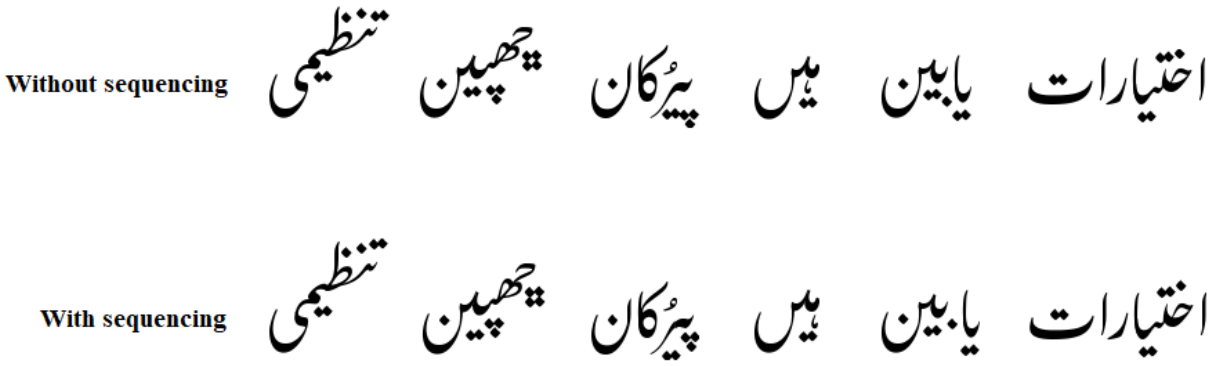
Region 4. We do not want nuqtas to form a straight line since that can cause visual confusion. In Region 4, the result is worse the closer the glyph is to being aligned with the double nuqta. The height of Region 4 is indicated by `sequence.valign.height` and the cost is weighted by `sequence.valign.weight`. It is always relative to the center of the neighboring glyph.

Region 5. This area is also controlled by `sequence.valign.height/weight`. Notice that at the lower edge of Region 5, the cost associated with Region 3 comes into play, because the target glyph is drifting from one undesirable position to another. Which is considered worse can be controlled by the respective `weight` attributes.

Region 6. This is where collisions occur between the single nuqta and the neighboring nuqta and is governed by the standard collision algorithm. The size of the rectangle is indicated by `collision.margin` on the neighboring glyph.

The wavy edges of the diagram indicate that the regions extend horizontally to the edges of the cluster indicated by the start- and end-flags, and vertically to infinity.

Here are some examples of the effects of sequencing.



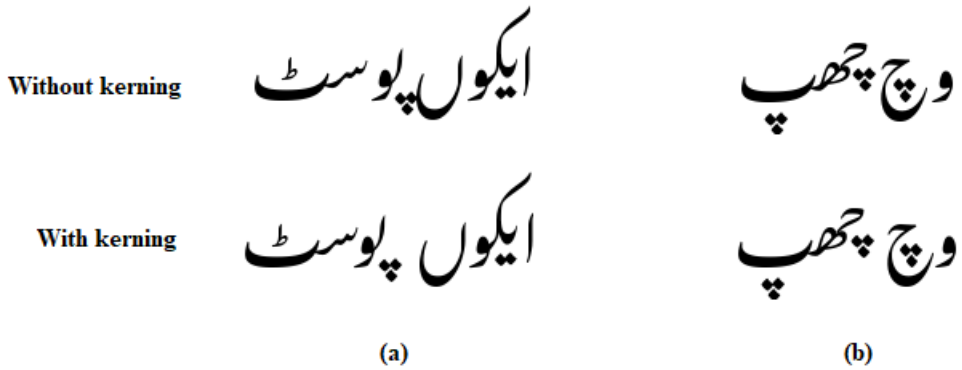
6.8.2 Kerning

Setting `AutoKern = true` in a pass's directives causes the automatic kerning algorithm to be run. *As well, `CollisionFix` must be set to something greater than zero for kerning to be achieved,* even if no shifting is needed.

The kerning mechanism applies "positive kerning," adding space after a base glyph, to avoid a collision with the following glyph. It will automatically take into account any glyphs such as nuqtas or diacritics that are attached to the base glyphs. To indicate that a glyph should be kerned, set the KERN flag in the `collision.flags` attribute. The `collision.margin` attribute indicates how much space should be left. Note that the START and END flags do not affect kerning, since it is assumed that kerning may very well need to take place over an intervening space.

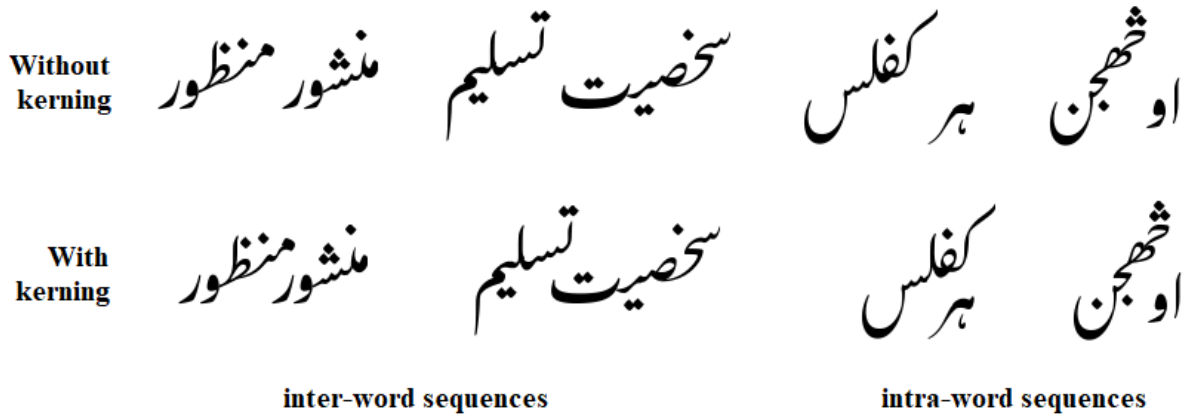
The kerning algorithm is also smart about intervening space characters, and will ensure that the kerning includes both what is necessary to fix the collision as well as the width of the space itself (even if there is no actual collision when the space is present). The IS-SPACE flag can be used to mark any glyph that should be treated this way (such as a visible representation of a space character).

The figure below shows examples. In example (a), kerning space is added to avoid the collision between the noon ghunna and the triple nuqat of the peh (which completely fills the space between the two words and more). In example (b), although there is no collision between the two words, the triple nuqat would still obscure the presence of the space, and the kerning algorithm adds horizontal space to reflect it.



In addition to avoiding collisions, the kerning algorithm can be used to create the desired amount of space between finals and initials, which maybe involve both negative and positive kerning. This can be seen most clearly in example (a) in the spacing between the final waw and initial seen.

More dramatically, negative or inward kerning can be used to create overlaps between diagonal sequences as is characteristic of the Nastaliq style. This can be achieved with both intra-word sequences and separate words. Again, the presence of any spaces marked IS-SPACE are detected and factored into the kerning. The `collision.min.x` attribute controls the amount of negative kerning permitted to achieve the overlap.



The `collision.max.y` and `collision.min.y` attributes are ignored for kerning.

6.8.3 Example

The following is a simplified example of how to use the automatic collision avoidance mechanism to handle collisions and kerning in Nastaliq-style Arabic.

```
// GDL-defined constants:
#define COLL_FIX       1
#define COLL_IGNORE    2
#define COLL_START     4
#define COLL_END       8
#define COLL_KERN     16
#define COLL_ISSPACE 128

#define ORDER_LEFTDOWN 1
#define ORDER_NOABOVE  4
#define ORDER_NOBELOW  8

// Macros for setting flags:
#define setbits(f,m,v) (f & (~m)) | v
#define SET_FIX(f)     setbits (f, COLL_FIX, COLL_FIX)
#define SET_START(f)   setbits(f, COLL_START, COLL_START)
#define SET_END(f)     setbits(f, COLL_END, COLL_END)
#define SET_KERN(f)    setbits(f, COLL_KERN, COLL_KERN)
#define SET_ISSPACE(f) setbits(f, COLL_ISSPACE, COLL_ISSPACE)

// My sequence classes:
#define SEQ_UNUQTA 1
#define SEQ_LNUQTA 2
#define SEQ_UDIAC  3
#define SEQ_LDIAC  4

table(glyph)

   // Shifting
```

```
    clsUpperNuqta = (gUpN1, gUpN2, gUpN3)
          // Upper nuqtas can be shifted up more than down:
          { collision {min.x=-300m; max.x=300m; min.y=-200m; max.y=500m};
            sequence.class = SEQ_UNUQTA}};

    clsLowerNuqta = (gLowN1,gLowN2,gLowN3
          // Lower nuqtas can be shifted down more than up:
          { collision {min.x=-300m; max.x=300m; min.y=-500m; max.y=200m};
            sequence.class = SEQ_LNUQTA}};

    // The sequencing is the same for upper and lower nuqtas—they should
    // flow in a top-right-to-bottom-left sequence:
    clsNuqta = (clsUpperNuqta, clsLowerNuqta)
       { collision {margin = 150m; marginweight = 200};
         sequence {
             order = ORDER_LEFTDOWN;
             above  {xoffset = 100m; weight = 300};
             below  {xlimit  = -50m; weight = 100};
             valign {height  = 150m; weight = 500} }};

    clsUpperDiac = (gFatha, gDamma)
       {collision {
             margin = 200m; marginweight = 200;
             min.x = -300m; max.x = 300m; min.y = -300m; max.y = 800m}};
       {sequence {
             // Keep upper diacritics above upper nuqtas:
             class = SEQ_UDIAC; order = ORDER_NOBELOW;
             proxClass = SEQ_UNUQTA}};

    clsLowerDiac = (gKasra)
       {collision {
             margin = 200m; marginweight = 200;
             min.x = -300m; max.x = 300m; min.y = -800m; max.y = 300m}};
       {sequence {
             // Keep lower diacritics below lower nuqtas:
             class = SEQ_LDIAC; order = ORDER_NOABOVE;
             proxClass = SEQ_LNUQTA}};

    // Kerning

    clsFinalIsolate = (clsFinal, clsIsolate)
          {collision {margin = 200m; min.x = -1000m; max.x = 5000m}};

endtable;

table(pos)

pass(1)

    // rules to attach nuqtas

endpass;

pass(2) {CollisionFix = 3}

    // Spaces serve as boundaries of ranges to check for collisions:
    gSpace { collision.flags = SET_START(collision.flags);
          collision.flags = SET_END(collision.flags) };

    // Fix nuqtas on this pass:
    clsNuqta { collision.flags = SET_FIX(collision.flags) };

endpass;
```

```
pass(3)
   // rules to attach diacritics
endpass;
pass(4) {CollisionFix = 3; AutoKern = 1}
   // Fix diacritics on this pass:
   clsUpperDiac { collision.flags = SET_FIX(collision.flags) };
   clsLowerDiac { collision.flags = SET_FIX(collision.flags) };
   // Also do kerning on this pass:
   clsFinalIsolate {
      collision.flags = SET_FIX(collision.flags);
      collision.flags = SET_KERN(collision.flags) };
   // Not necessary, since an actual space is already treated
   // as whitespace:
   gSpace {collision.flags = SET_ISSPACE(collision.flags)};
endpass;
endtable;
```

6.8.3.1 Discussion

The general approach we take is to first attach nuqtas and fix their collisions, then attach diacritics and fix their collisions. Kerning is left as the last step; it can be treated as a last resort since it is always possible to kern in order to avoid a collision between sequences, and it should never create any new collisions.

We set most of the `collision` and `sequence` attributes as glyph attributes, from which the slot attributes are initialized. It would be possible to adjust their values within rules, as needed. Although `collision.flags` exists as a glyph attribute, it seems to make more intuitive sense to set it within a rule, at the place in the process where we want the fix to happen.

Clusters for testing collisions are delineated by spaces.

Note that the first group of `#define` statements are constants that are defined within the Graphite system. The set that start with `SEQ` are defined by the GDL programmer.

6.8.3.1.1 *Possible enhancements*

If desired, collision avoidance can be turned off for nuqtas on the final pass, using:

```
#define CLEAR_FIX(f) setbits (f, COLL_FIX, 0)
```

and in pass(4):

```
clsNuqta { collision.flags = CLEAR_FIX(collision.flags) };
```

It may be convenient for the purposes of debugging (for instance, when using the Graide development tool) to separate the rules setting the collision parameters from the running of the collision algorithm itself—especially if the rules are quite complex. This can be done easily, e.g.:

```
pass(4)
   // Fix diacritics on this pass:
   clsUpperDiac { collision.flags = SET_FIX(collision.flags) };
   clsLowerDiac { collision.flags = SET_FIX(collision.flags) };
   // Also do kerning on this pass:
   clsFinalIsolate {
      collision.flags = SET_FIX(collision.flags)
      collision.flags = SET_KERN(collision.flags) };
```

```
endpass;
pass(5) {CollisionFix = 3; AutoKern = 1}
   // No rules, just run the collision avoidance algorithm
endpass;
```

6.9 Backward-compatible feature IDs

In the past, Graphite font features were geneally given mnemonic IDs such as ‘litr’, ‘viet’, and ‘dig4’. Now that OpenType supports character variants and stylistic sets, you may want Graphite font tables to use IDs that are compatible with OpenType’s, such as ‘ss01’ and ‘cv24’. However, it might not be adequate to simply remove the old IDs from the font since that would break backward compatibility.

The solution is to assign multiple IDs to a GDL feature definition. This in effect creates multiple features with identical behavior in the font. The new ID will be offered to the user in UI mechanisms; the old ID is not offered but is still effective for existing data that might use it.

Implementing this solution in a Graphite font has two main aspects.

6.9.1 GDL feature definition

A feature definition can be extended to include multiple IDs. All but the first ID must be marked “hidden.”

```
featureGdlName {
      id = “cvXX”;      // ie, an ID that is compatible with OpenType
      id.hidden = “prev”;     // a mnemonic ID
      default = default_value;
      etc.
};
```

Note that using a “hidden” ID can be used to simply create a feature with a single ID that is not shown in the user interface but may still be accessed by those who know the “secret code.”

6.9.2 Feature testing

As indicated above, including a hidden ID actually causes the resulting Graphite font to have an extra feature with that ID. This means that the GDL code that tests a feature must explicitly test all the features that result from the single GDL definition—the public feature as well as any hidden features.

A special syntax is available to test hidden features, consisting of the GDL name followed by ‘__’ and then the hidden ID.

So if your original GDL looked like

```
if (featureGdlName == 2)
```

the new code would be

```
if (featureGdlName == 2 || featureGdlName__prev == 2)
```

The ID can also be included for public features, so the following is also valid:

```
if (featureGdlName__cvXX == 2 || featureGdlName__prev == 2)
```

6.9.2.1 Testing default values

Special care must be taken when testing default values. For a non-default value, if either the public feature or the hidden feature is set to that value, the feature will be considered activated. But notice that the other feature will (most likely) still have the default value. This means that only if both the public

feature and the hidden feature have the default value can the feature be considered to use the default setting. For this reason, default features should generally be tested using a logical AND operation rather than OR:

```
if (featureGdlName == 0 && featureGdlName_prev == 0)
// rules to run in the default case
```

6.9.2.2 Incompatible values

This raises the question of how to handle incompatible values. For instance, what if cvXX = 2 and prev = 1? This could happen if the user has some existing data marked with prev = 1 and then uses the UI mechanisms to set cvXX to 2.

Here is a GDL test that takes this possibility into account:

```
if (featureGdlName == 0
      && (featureGdlName_prev == 0 && featureGdlName == 0))
// rules to run in the default case
```

In other words, the hidden feature is only taken into account when the public feature is set to the default.

6.9.3 Example

An older Graphite font has a feature controlling which form of the uppercase eng is used. Here is how the updated code would look like to handle an ID that matches an OpenType feature.

```
table(feature)

eng {
      id = “cv43”;
      id.hidden = "Engs";
      name.1033 = string("Uppercase Eng alternates");
      default = descender;
      settings {
            descender {
                  value = 0;
                  name.1033 = string("Large eng with descender");
            }
            base {
                  value = 1;
                  name.1033 = string("Large eng on baseline");
            }
            capital {
                  value = 2;
                  name.1033 = string("Capital N with tail");
            }
            short {
                  value = 3;
                  name.1033 = string("Large eng with short stem");
            }
      }
}

endtable;


table(substitution)

if (eng == descender && eng__Engs == descender)
// rules to produce large eng with descender
endif;
```

```
if (eng == base || (eng__Engs == base && eng__cv43 == descender))
// rules to produce large end on baseline
endif;

if (eng == capital || (eng__Engs == capital && eng__cv43 == descender))
// rules to produce capital N with tail
endif;

if (eng == short || (eng__Engs == short && eng__cv43 == descender))
// rules to produce large eng with short stem
endif;

endtable;
```

7 Reference

7.1 Attributes

This section provides a reference summary of each of the different types of attributes that are currently available in Graphite (excluding glyph metrics). Attributes only have meaning in certain contexts. These contexts are within a particular table type or as a glyph attribute which is passed to a slot.

7.1.1 Named numerical glyph attributes

The user can define arbitrary glyph attributes in the glyph table. They consist of a name assigned a numeric value. The name can then be read later in any table for any purpose the author desires.

7.1.2 advance (adv)

The `advance` slot attribute is only applicable in the positioning table. It specifies the distance between the origins of two glyphs. `Advance.x` and `advance.y` are set or read independently of each other. By default the `advance` of a slot is equal to the advance of the glyph in the slot (`advanceheight` and `advancewidth` glyph metrics). It controls the positioning of the glyph to the right of the glyph it is set on. Increasing the `adv.x` value will move the next glyph's position to the right. A typical GDL statement would use the += or -= operators to do this is, for instance:

```
glyphToAdjust {adv.x += 100m}
```

All glyphs following on the same line will be moved relative to their screen position. The `adv` and `shift` attributes are used together for kerning.

7.1.3 attach (att)

All the `attach` attributes are slot attributes and are only applicable within a positioning table. There are four sub-attributes of `attach` and they are used to indicate how two glyphs (or glyph slots) are positionally attached to each other.

7.1.3.1 attachment points

Several types of named glyph attributes have special support. For creating attachment points, three functions exist: `point`, `gpath`, `gpoint`. Setting a name equal to one of these functions creates a set of glyph attributes. (See the Advanced Concepts section.) The set of attributes so created will then be implicitly used with the `attach.at` and `attach.with` slot attributes. Attachment points are created in the glyph table. The name of a particular attachment point is private to a particular glyph, thus different glyphs may have the same names for their attachment points.

7.1.3.2 attach.to

This is a slot reference to another slot in the form `@`*`n`* where *`n`* is the context reference indicating which slot this slot is to be attached to. The process of attaching two slots sets the `insert` slot attribute to false by default.

7.1.3.3 attach.at

This slot attribute specifies the name of the attachment point on the glyph that this glyph is attaching to. This attribute works in conjunction with `attach.with` to provide relative positioning of the two glyphs. Notice that the point named in this attribute is on the glyph that doesn't move. The point name is a named glyph attribute defined in the glyph table.

7.1.3.4 attach.with

This slot attribute specifies the name of the attachment point on this glyph which is positioned over the `attach.at` attachment point on the other glyph.

7.1.3.5 attach.level

This slot attribute specifies the level number of the attachment for purposes of calculating composite metrics. The level number can be used as a dotted postfix with glyph metrics. When accessing composite metrics for a group of glyphs attached together all attachments at the specified level or lower are used.

7.1.4 breakweight (break)

The `breakweight` attribute of a glyph is set in the glyph table or by default. The `breakweight` attribute of a slot is determined from the glyph in that slot, but may also be set in the line-breaking table to provide contextual line-breaking information to the line-breaking algorithm. The lower the value of `breakweight`, above 0, the higher the priority of the line-break. This attribute can also be read from the line-break pseudo-character to determine its actual break weight.

7.1.5 collision

The `collision` attributes are used to control the behavior of automatic collision avoidance mechanism. All of them are both glyph and slot attributes except `collision.complexFit which` is only a glyph attribute.

7.1.5.1 collision.complexFit

This glyph attribute indicates that the shape of the glyph is such that the collision algorithm needs a finer-grained estimate of the glyph shape. This is useful for glyphs with concave edges.

7.1.5.2 collision.exclude.glyph, collision.exclude.offset.x/y

The `collision.exclude.glyph` attribute provides a way to expand the shape of the glyph that is used to avoid collisions. The value of the attribute is a glyph ID, and that glyph is added to the shape of the main glyph for use by the collision avoidance algorithm. The offset attributes allow the `exclude.glyph` to be positioned appropriately relatsive to the main glyph.

7.1.5.3 collision.flags

This attribute indicates how the glyph should be treated by the collision avoidance algorithm. The value is a bitmap containing the following values:

- FREEZE = 0: don't move this glyph.
- FIX = 1: this glyph should be moved by the algorithm if necessary.
- IGNORE = 2: this glyph should be ignored by the algorithm.
- START = 4: this glyph marks the beginning of a sequence whose glyphs may collide (generally the first glyph after a space).
- END = 8: this glyphs marks the end of a sequences whose glyphs may collide (generally the last glyph before a space).
- KERN = 16: this glyph should be kerned if necessary to avoid colliding with a following glyph (generally the last glyph of a contextual sequence).
- IS-SPACE = 128: this glyph is white space (or a visible glyph that should be treated as white space).

The `setbits` function is a good way to set and clear these bits; e.g.:

```
#define setbits(f,m,v) (f & (~m)) | v
#define SET_IGNORE(f)    setbits(f, 2, 2)
#define CLEAR_IGNORE(f)  setbits(f, 2, 0)
```

7.1.5.4 collision.margin, collision.marginweight

The `collision.margin` attribute indicates how much margin should be kept between the glyph and surrounding glyphs. The `collision.marginweight` attribute indicates how much the algorithm should penalize a violation of the margin, compared to other costs. These can be defined as either a glyph or slot attributes.

7.1.5.5 collision.min.x, collision.min.y, collision.max.x, collision.max.y

Glyphs that can be shifted by the collision algorithm have limiting rectangle in which they can move. These attributes indicate the movement limits for the glyph. These can be defined as either a glyph or slot attributes.

Unlike the `shift.x` attribute, for `collision.max.x` and `collision.min.x`, positive values always refer to right-ward movement and negative values to left-ward movement, regardless of the script direction.

The `max.x` and `min.x` attributes also indicate how much kerning can be applied; `max.y` and `min.y` are not relevant for kerning.

7.1.6 component (comp) *[Not implemented]*

Ligatures may be described as having components. A component-structured ligature allows the cursor to be placed within the ligature and allows association between the components of the ligature and underlying codepoints. Each ligature component has a glyph attribute name which is private to the glyph in question and is arbitrary. In the following descriptions, `<name>` will be used to mark where a component name would be placed.

Note: ligature components have not been implemented in the Graphite2 engine (as of version 1.3.12).

7.1.6.1 component.<name>.box

This is a glyph attribute and should only be set in the glyph table. It is meaningless in all other table types. Each ligature component has a bounding box which is a list of 4 values.

For creating ligature components, the `box` function exists. It will create a set of glyph attributes when assigned to the `component.<name>` attribute. These attributes will be implicitly associated with the component.<name>.reference slot attribute.

7.1.6.2 component.<name>.reference (ref)

This is a slot attribute used in substitution tables to associate an underlying codepoint with a surface glyph component. Thus ligature components must go through the same cursor tracking that any other slot would. In other words, ligature components are very much like individual slots for cursor tracking purposes.

7.1.7 directionality (dir)

This attribute can be set for glyphs in the glyph table and slots in the substitution table. It can only be set on glyphs that either do not have associated Unicode IDs in the cmap (including pseudo glyphs) or which correspond to Unicode IDs in the PUA. Glyphs associated with Unicode IDs use the standard directionality for that codepoint. It is used by the directionality algorithm to arrive at a glyph order for rendering.

7.1.8 insert

This slot attribute used in either the substitution or positioning table indicates whether a glyph can have the cursor placed before it. By default this attribute is set to 0 (false) on attachment, but there are situations where this is not the required behavior. In such a situation, `insert = 1` (true) is used to indicate that the cursor can be placed before the glyph. This is used for side attachment as in nastaliq.

7.1.9 justify

The `justify` attributes are used to accomplish justification—stretching or shrinking a line of text to fit within a given amount of space. In the current version of Graphite, the level indicator is optional; for instance, either justify.0.stretch or justify.stretch may be used (as of version 1.3.12 of the Graphite2 engine).

7.1.9.1 justify.shrink

This attribute indicates the maximum amount by which the glyph can be shrunk. It is both a glyph and slot attribute. The value is in em-units.

7.1.9.2 justify.step

This attribute indicates the step or "chunk" by which the glyph can be stretched or shrunk. A positive value relates to stretching and a negative value relates to shrinking. It is both a glyph and slot attribute. The value is in em-units.

7.1.9.3 justify.stretch

This attribute indicates the maximum amount by which the glyph can be stretched. It is both a glyph and slot attribute. The value is in em-units.

7.1.9.4 justify.weight

This attribute indicates the preference that the justification algorithm should give to stretching this glyph. It is both a glyph and slot attribute. The default value is 1 and the maximum value is 255.

7.1.9.5 justify.width

This slot attribute indicates the amount of stretching (positive value) or shrinking (negative value) that has been assigned to this glyph by the justification algorithm. The value is in em-units.

7.1.10 kern

Kerning is a slot attribute used in the positioning table and is implemented by applying `shift` and `advance`. It cannot be read.

Note that this attribute has nothing to do with the kerning that happens during the automatic collision avoidance algorithm.

7.1.11 position (pos)

The `position` (or `pos`) slot attribute allows one to determine the distance between two glyphs. It is only readable in the substitution or positioning table. Both `pos.x` and `pos.y` exist. `Pos.y` provides the distance of a glyph's upper left corner from the baseline. Note that the value of any single `pos.x` value is not meaningful; only the difference between two `pos.x` values is useful.

7.1.12 sequence

The `sequence` attributes can be used to achieve the characteristic behavior of Nastaliq-style script. They constrain how the glyphs are moved relative to each other, and can be used to enforce a visual sequencing effect.

7.1.12.1 sequence.class

Glyphs that are handled similarly and positioned relative to each other should be put in the same class. The classes are indicated by arbitrary integers. For instance, you might put all the upper nuqtas in class 1, the lower nuqtas in class 2, etc.

These attributes all exist as both glyph attributes and slot attributes.

7.1.12.2 sequence.order

The `order` attribute indicates how glyphs should be ordered relative to other glyphs in the same class, or the glyphs in the "proximate class" (`sequence.proxClass`). The value is a bitmap with the following flags:

- NONE = 0: don't enforce any order relative to other glyphs.
- LEFTDOWN = 1: keep subsequent glyphs in the same class positioned to the left and down from this glyph.
- RIGHTUP = 2: keep subsequent glyphs positioned to the right and up from this glyph. (Note: this value is included for completeness; it has no known real-life application.)
- NOABOVE = 4: prevent this glyph from being positioned above glyphs in the `proxClass`. For instance, the following code might be used to set the attribute on a lower diacritic to keep it below lower nuqtas:

```
#define LNUQTA 1  // arbitrary
#define LDIAC 2   // arbitrary
#define NOABOVE 4
c_lowerDiac {sequence {class = LDIAC; proxClass = LNUQTAS;
      order = NOABOVE;}}
```

- NOBELOW = 8: prevent this glyph from being positioned below glyphs in the `proxClass`. For instance, it can used to force upper diacritics to be kept above upper nuqtas.

7.1.12.3 sequence.proxClass

Sometimes glyphs of one class need to be positioned relative to the glyphs of a different class. The `proxClass` attribute indicates which "proximate class" should be considered. For instance, you could set the `sequence.proxClass` attribute of an upper diacritic to the class containing the upper nuqtas, as well as setting `sequence.order` to 8 (NOBELOW).

7.1.12.4 sequence.above.xoffset/weight

These attributes are used by glyphs that have `sequence.order` set to LEFTDOWN.

The `sequence.above.xoffset` attribute specifies how close a neighboring sequenced glyph positioned above can approach horizontally to the margin of the glyph in question, where the margin is specified by `collision.margin`. (IS IT TO THE MARGIN OR TO THE BOUNDING BOX?)

The `weight` attribute indicates how costly encroachment into this horizontal space should be considered, compared to other violations (`collision.marginweight`, `sequence.below/valign.weight`, etc.) The further into this space the neighboring glyph would go, the higher the cost.

7.1.12.5 sequence.below.xlimit/weight

These attributes are used by glyphs that have `sequence.order` set to LEFTDOWN.

The `sequence.below.xlimit` attribute specifies how close a neighboring sequencing glyph that is vertically below the glyph in question can approach horizontally to the margin of that glyph. (Note that

a previous glyph that is positioned below is already in an infelicitous relationship to the glyph in question.)

The `sequence.below.weight` attribute takes into account the amount the neighboring question is moved, relative to other costs (`collision.marginweight`, `sequence.above/valign.weight`, etc.)

7.1.12.6 sequence.valign.height/weight

These attributes are used by glyphs that have `sequence.order` set to LEFTDOWN.

The `sequence.valign.height` attribute indicates how far a neighboring sequenced glyph should be vertically offset from the glyph in question. This prevents a sequence of glyphs from aligning vertically and forming a straight horizontal line.

The `sequence.valign.weight` attribute indicates how costly alignment violations should be considered, compared to other violations (`collision.marginweight`, `sequence.above/below.weight`, etc.)

7.1.13 shift

The `shift` slot attribute is used in the positioning table. It displaces a glyph from its normal position without altering the screen position of any other glyph. `Shift.x` and `shift.y` may be set and read independently of each other.

7.1.14 metrics

Glyph metrics are available in all tables as read only values. A previous section lists all available metrics.

7.1.15 user

There are sixteen user-definable slot attributes with the names `user1`, `user2`, `user3`, …, `user16`. See the description of these in the Advanced Concepts section.

7.2 Attribute Table

Below is a table of all attributes along with the GDL table(s) they can be used in. Rules in subsequent tables can query attributes that are usable by previous tables.

Glyph Attributes **Glyph Table**	
`breakweight`	`mirror {glyph; isEncoded}`

`collision {flags;`
`  min {x; y}; max {x; y};`
`  margin; marginweight;`
	metrics (read only)

`component.<name> (box)`	named number
`directionality`	named points (`gpoint`, `point`, `gpath`)

`justify.<level>`[1] `{stretch; shrink;`
`  step; weight}`
	`sequence {class; proxClass; order;`
	`  above {xoffset; weight};`
	`  below {xlimit; weight};`
	`  valign {height; weight}}`

Slot Attributes	
Linebreak Table	
`breakweight`	`user1, user2,` etc.
Substitution Table	
`component.<name>.reference`	`justify.<level>`[1] `{stretch; shrink;` `  step; weight}`
`directionality`	`position {x; y}` (read only)
`insert`	`user1, user2,` etc.
Justification Table	
`justify.<level>`[1] `{stretch; shrink;` `  step; weight}` (read only)	`justify.<level>`[1]`.width`
Positioning Table	
`advance {x; y}`	`justify.<level>`[1]`.width`
`attach {to; at; with; level}`	`position` (read-only)

`collision {flags;`
`  min {x; y}; max {x; y};`
`  margin; marginweight;`
`  exclude {glyph; offset {x; y}}}`
	`sequence {class; proxClass; order;`
	`  above {xoffset; weight};`
	`  below {xlimit; weight};`
	`  valign {height; weight}}`

`insert`	`shift {x; y}`
`kern {x; y}` (write only)	`user1, user2,…,user16`

[1] In the current version of Graphite2 (1.3.12), the level indictor is optional; i.e., `justify.stretch`, `justify.weight`, etc. are also valid syntax.

justify.<level>[1] {stretch; shrink; step; weight} (read only)	

7.3 Abbreviations

The following table lists all abbreviations available in GDL. These abbreviations are defined by #including the “stddef.gdh” file.

advance	adv		environment	env
advanceheight	ah		justification	just
advancewidth	aw		linebreak	lb
attach	att		leftsidebearing	lsb
boundingbox	bb		positioning	pos
breakweight	break		reference	ref
component	comp		rightsidebearing	rsb
directionality	dir		substitution	sub, subs
endenvironment	endenv			

8 Language Structure

To help with implementation and a conceptual understanding of the description language, we examine here the description language as a general computer language.

The first important concept is that the language is non-procedural. Thus all procedural elements should be understood in this light. Functions should only be used as ways of getting at atomic values and should have no side effects. The selection mechanisms (`if()` and `pass()`) should be understood as such, as ways of selecting rules.

It is anticipated that a description file will be processed using a two stage compiler: parser and compiler. For this reason the language has been designed to be as generic as possible at the surface syntax, and to have as much of the particularization passed down to the semantic level where the compiler can deal with it. Thus, all functions are resolved by the compiler rather than the parser.

8.1.1 Primitive Types

At the lowest level there are a few basic types and these have been minimized and made as ubiquitous as possible.

8.1.1.1 Case Sensitivity

User-defined names (classes, features, and glyph attributes) are case sensitive and must use only 7 bit ASCII characters. Keywords, such as `table`, are case insensitive and should never be used as a user-defined name. Slot attributes and glyph identification functions are case sensitive and must be in lower case. The global settings and directives must be in mixed case as specified in this document.

8.1.1.2 Number

One primitive is the number. By default a number is a number however it is expressed. For positioning information, it is necessary to allow a number to be scaled based on the value of the `MUnits` directive. Scaled numbers are indicated by postfixing an 'm'.

8.1.1.2.1 *Reference*

This variant of number can be thought of as another type of units. References occur within the context of a rule and are used to indicate that the value to be used should be resolved to a location in the string or glyph stream rather than as simply a number. References are preceded by "`@`."

8.1.1.3 String

In GDL, strings contain 8-bit characters, possibly with an associated codepage. Internally strings are converted to Unicode.

8.1.1.4 List

Lists can consist of any other primitive type. There has been nothing in the language which requires that lists should be able to nest. Apart from this, lists are untyped.

8.1.2 Implicit Glyph Attributes

Using `gpath`, `gpoint`, `point`, or `box` to create attachment points and ligature component boxes is really shorthand for specifying several attributes at once. The shorthand form is not required. Here is a table specifying the equivalences.

The following shorthand …	… is equivalent to:
`<name> = gpath(<num>)`	`<name>.gpath = <num>`

	<name>.xoffset = 0 <name>.yoffset = 0
<name> = gpath(<num>, <xoffset>, <yoffset>)	<name>.gpath = <num> <name>.xoffset = <xoffset> <name>.yoffset = <yoffset>
<name> = gpoint(<num>)	<name>.gpoint = <num> <name>.xoffset = 0 <name>.yoffset = 0
<name> = gpoint(<num>, <xoffset>, <yoffset>)	<name>.gpoint = <num> <name>.xoffset = <xoffset> <name>.yoffset = <yoffset>
<name> = point(<x>, <y>)	<name>.x = <x> <name>.y = <y> <name>.xoffset = 0 <name>.yoffset = 0
<name> = point(<x>, <y>, <xoffset>, <yoffset>)	<name>.x = <x> <name>.y = <y> <name>.xoffset = <xoffset> <name>.yoffset = <yoffset>
comp.<name> = box(<xmin>, <ymin>, <xmax>, <ymax>)	comp.<name>.left = <xmin> comp.<name>.bottom = <ymin> comp.<name>.right = <xmax> comp.<name>.top = <ymax>

For example, the following code:

```
table(glyph);
   gA = unicode(0x0041) {udap = gpath(3)};
   gB = unicode(0x0301) {lap = point(adv.width / 2, bb.bottom)};
endtable;
```

is equivalent to:

```
table (glyph);
   gA = unicode(0x0041);
   gA.udap.gpath = 3;
   gA.udap.xoffset = 0;
   gA.udap.yoffset = 0;
   gB = unicode(0x0301);
   gB.lap.x = adv.width / 2;
   gB.lap.y = bb.bottom;
   gB.lap.xoffset = 0;
   gB.lap.yoffset = 0;
endtable;
```

9 Glossary

advance height – the amount by which the current display position is adjusted vertically after rendering a given glyph. This number is generally only meaningful for vertical writing systems, and is usually zero within fonts used for horizontal writing systems.

advance width – the amount by which the current display position is adjusted horizontally after rendering a given glyph.

ASCII – a standard that defines the 7-bit numbers (codepoints) needed for the U.S. English writing system. (American Standard Code for Information Interchange)

ascent – the distance between the top of the line of text and the baseline, as defined within a font.

baseline – the vertical point of origin for all the glyphs rendered on a single line. Roman scripts have a baseline on which the glyphs appear to "sit," with occasional descenders below. Many Indic scripts have a "hanging" baseline, in which the bulk of the letters are placed below the baseline, with occasional ascenders above the line.

bidirectionality – the characteristic of some writing systems to contain ranges of text that are written left-to-right as well as right-to-left. Specifically, in Arabic and Hebrew scripts, most text is written right-to-left, but numbers are written left-to-right.

bounding box – the rectangular area containing the entire visual portion of a glyph.

character – an abstract symbol used in writing, and the most fundamental unit of data representation.

cmap – character-glyph map: the table within a font containing a mapping of codepoints (characters) to glyph ID numbers. In a Unicode-based font the codepoints are Unicode values; in other fonts they may correspond to other encodings.

codepage – a mapping between a set of 8-bit or double-byte codepoints and corresponding Unicode codepoints. Each codepage has an identifying number used to access the mapping in system functions; for example, the default Roman codepage for Western European languages is codepage 1252.

codepoint – a number that represents a character. For instance, in Unicode and ASCII standards, the number 97 is used to represent the lowercase 'a'.

descent – the distance between the bottom of the line of text and the baseline, as defined within a font.

diacritic – a mark attached to another character to modify it in some way.

em square – the square grid which is the basis for the design of all glyphs within a given font; so called because it historically corresponded to the size of the letter M. When rendering, the requested point size specifies the size of the font's em square to which all glyphs are scaled.

em units – the number of units defined in a font's em square. All coordinates for points in a glyph are defined using em units.

font – a file containing a collection of glyphs used together to render text.

glyph – a shape that is the visual representation of a character. Different fonts will have slightly different shapes representing the same character. For instance, **a**, **a**, and *a* are all glyphs that correspond to the character 'a'.

glyph ID – the unique number within a font identifying a single glyph.

kern – to adjust the display position while rendering in order to visually improve the spacing between two glyphs. For instance, kerning causes the word "WAVE" to be rendered as "WAVE", reducing the

illusion of white space between the diagonal strokes of the W, A, and V. In terms of Graphite slot attributes, kern is an adjustment of both shift and advance: the origin of the glyph is changed, and the display position is adjusted by an equal amount after the glyph is rendered.

left-side bearing – the white space at the left edge of a glyph's visual representation, or more specifically, the distance between the current horizontal display position and the left edge of the glyph's bounding box. A positive left-side bearing indicates white space between the glyph and the previous one; a negative left-side bearing indicates overlap or overhang between them.

ligature – a single shape or glyph that represents two or more underlying characters.

Postscript name – a name associated with a glyph by the font's designer; originally a name assigned by Adobe to certain standard glyphs.

Private Use Area (PUA) – a range of Unicode codepoints (E000 – F8FF and planes 15 and 16) that are reserved for private definition and use within an organization or corporation.

render – to display or draw text on an output device (computer screen, paper, etc.).

right-side bearing – the white space at the right edge of a glyph's visual representation, or more specifically, the distance between the display position after a glyph is rendered and the right edge of the glyph's bounding box. A positive right-side bearing indicates white space between the glyph and the following one; a negative right-side bearing indicates overlap or overhang between them.

script – a collection of characters and their basic behaviors that are mutually associated and identifiable, such as Roman, Arabic, Cyrillic, Chinese, etc.

side bearing – the white space at the edge of a glyph; see *left-side bearing, right-side bearing.*

Unicode – a comprehensive character-encoding standard intended to cover all the scripts of the world. In the Unicode standard, characters are typically encoded using 16-bit codepoints.

writing system – the subset of a script that is used by a particular language in a particular location or situation, characterized by rendering behavior, sorting, hyphenation conventions, etc. For example, English, German, and French all use Roman script, but have distinct writing systems. Mongolian can be written with two writing systems from two script families: Mongolian and Cyrillic. Ancient and modern Greek use different writing systems that are varieties of the same basic Greek script.

10 Appendix: The need for Graphite

The most immediate question that comes to mind when considering a new smart font description format is why the need for yet another description format? In answer to this question, we need to consider those already in existence and whether they meet our needs.

A general consideration to make when deciding whether to break with existing standards is to see what support exists for those standards and the difficulty of providing the support should it be lacking.

10.1 OpenType

The natural solution for the PC is to consider Microsoft's solution to the smart font rendering problem. This is OpenType, a set of tables which are added to a TrueType font to allow for glyph substitution, glyph positioning, multiple baselines and justification.

One of the stated principles of OpenType[1] is that writing system behavior should be handled in the application rather than in the font or operating system (despite this being against ISO recommendations[2]). This results in OpenType lacking in some areas. The particular area of concern is that OpenType does not support glyph reordering, a basic required mechanism. OpenType's features are also weak in that they are only boolean and are not named, only consisting of a tag.

10.1.1 Uniscribe

Uniscribe is Microsoft's layout engine, which was first shipped with Windows 2000. It will work in other 32 bit operating systems. It provides a programmer's API for smart script layout and rendering. It is built upon OpenType, and in addition to OpenType's capabilities, provides support for re-ordering and hit testing. Thus it removes some of the weaknesses inherent in OpenType.

Unfortunately, Uniscribe has not been built in an extensible fashion, so no new behaviors can be added or changed. (See *Graphite: An Extensible Rendering Engine for Complex Writing Systems* for a discussion of the need for extensible rendering capabilities.)

While Uniscribe is slated to support rendering of all of Unicode, it is not expected to provide any support for the Private Use Area. The PUA is an essential part of the strategy of defining encodings for non-standardized scripts and those that are under development, so lack of PUA support is a critical problem.

10.2 AAT

Apple Advanced Typography (formerly GX) is an existing format useable on the Mac by a few applications. Since it has been successfully used for a number of projects, it is difficult to say that it is severely lacking in any technical way. But, while the state machines are a powerful pattern matching mechanism, the actions available to any state machine are fairly weak, especially in the case of insertion. In addition, AAT has a number of implementation limitations especially regarding glyph insertion. This is not to say that any particular font could not be implemented in AAT, but that that implementation would be very different in design than for a rule based, higher level, linguistically motivated description.

---

[1] "As much as possible, the tables of the OpenType layout define only the information that is specific to the font layout. The tables do not try to encode information that remains constant within the conventions of a particular language or within the typography of a particular script. Such information that would be replicated across all fonts in a given language belongs in the text-processing application for that language, not in the fonts" (from OpenType Specification v1.2, November 1998; see www.microsoft.com/typography/tt/tt.htm).

[2] ISO/IEC JTC1/SC18/WG8 "PDTR15285. An operational model for characters and glyphs"

If AAT were chosen as a target technology, then we would have to implement a AAT engine for Windows. If we have to implement something, then we may as well take the opportunity of producing something more powerful and expressive. It is still intended to produce a AAT compiler, but this is a non-trivial activity, and may place some limitations on any given description.

Should either of these technologies, AAT or OpenType, become sufficiently ubiquitous and useable, then there is nothing to stop us changing direction to make use of them. But, at the time of writing, there is no foreseeable solution for Windows, either available or even promised. Therefore it seems wise to develop our own technology, while monitoring the industry and being ready to adjust accordingly.

10.3 SDF

An existing product within SIL has been developed to address the particular needs for context sensitive and cursive scripts. It provides good word positioning information and handles cursor tracking using Unicode codepoints. It only supports substitution without re-ordering and has no positioning support. As such it is a good start and is evidence of a pressing need for a sufficient solution for SIL applications.

Index

0

8

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z