Graphite Tutorial

Version 6.0

The following exercises are provided to give you an introduction to programming Graphite fonts, including an explanation of the fundamentals of the system and basic experience in using its most important smart rendering capabilities. This tutorial is not intended to be a comprehensive overview of the Graphite system, nor to provide an exhaustive discussion of all the features and syntax of the GDL programming language. For a complete discussion of GDL, see the "Graphite Description Language" document.

In order to use these tutorials, you will be using the Graide tool (Graide stands for GRAphite Interactive Development Environment). Graide allows you to run the Graphite compiler to create a Graphite enabled font, and test the results of the font using simple test data. It also includes debugging tools to analyze the behavior of your font.

If you choose not to try every exercise, you should put priority on the exercises that include a section called "Exploring Graide…".

When experimenting with Graide, this tutorial assumes that the GDL program you are working with looks very similar to the provided solution. If your program is quite different, it may be helpful to replace your program with the provided solution for the purposes of exploring the features of Graide.

Unit 1: Running, initializing and debugging with Graide

When you execute the Graide program with no arguments, the first step will be to create a configuration file indicating at minimum the following aspects of your project:

- Font (.TTF) file
- Graphite source code (.GDL) file

Other files that are used or created by Graide include:

- Configuration (.CFG) file
- Test data (.XML) file
- Debugger (.GDX) file
- Font compilation error message file (gdlerr.txt)

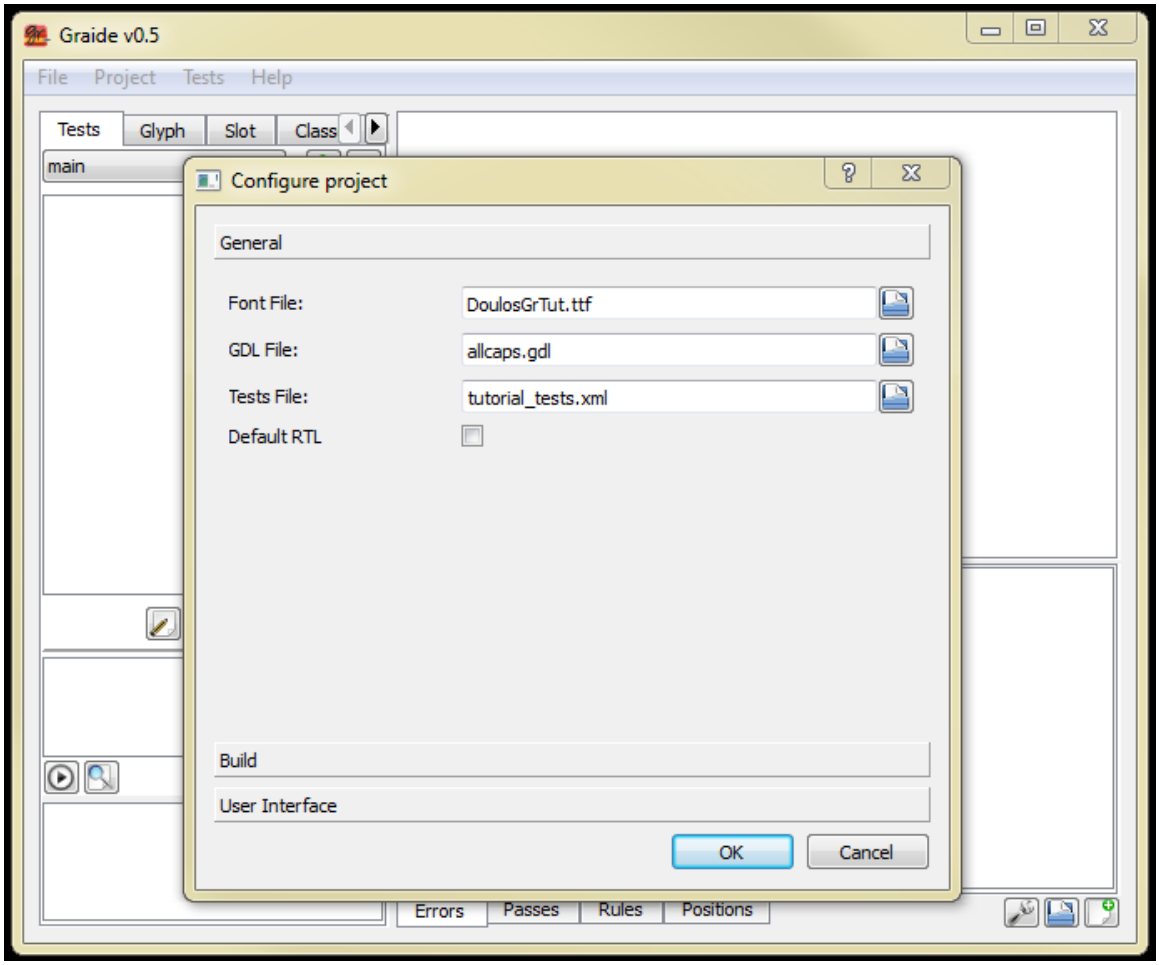
Exercise 1a

Step 1. Create or locate a workspace folder for this tutorial; this is where your fonts and source code files will go. Copy the files called "allcaps.gdl" and "DoulosGrTut.ttf" from the tutorial materials folder into that folder. Make sure the file properties are set to read-write.

Step 2. Run graide.exe. Choose Project → New Project. You will be asked for a configuration file. Navigate to your workspace folder and enter the file name as "tutorial.cfg". (Be sure to include the .cfg extension.)

Step 3. A dialog will appear requesting other project details.

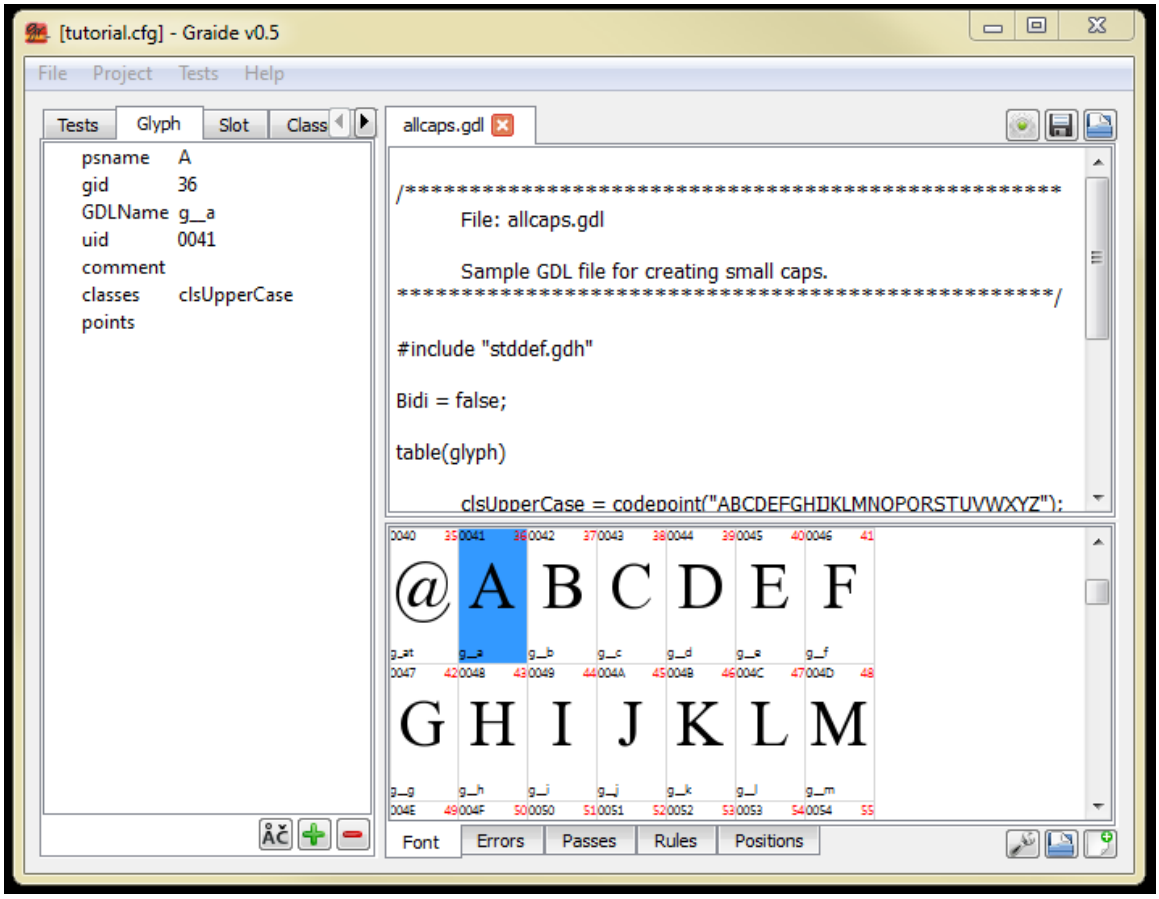
- To set the Font File, choose the DoulosGrTut.ttf file you copied in Step 1.
- To set the GDL File, choose the allcaps.gdl file you copied in Step 1.
- Set the Tests File to "tutorial_tests.xml".



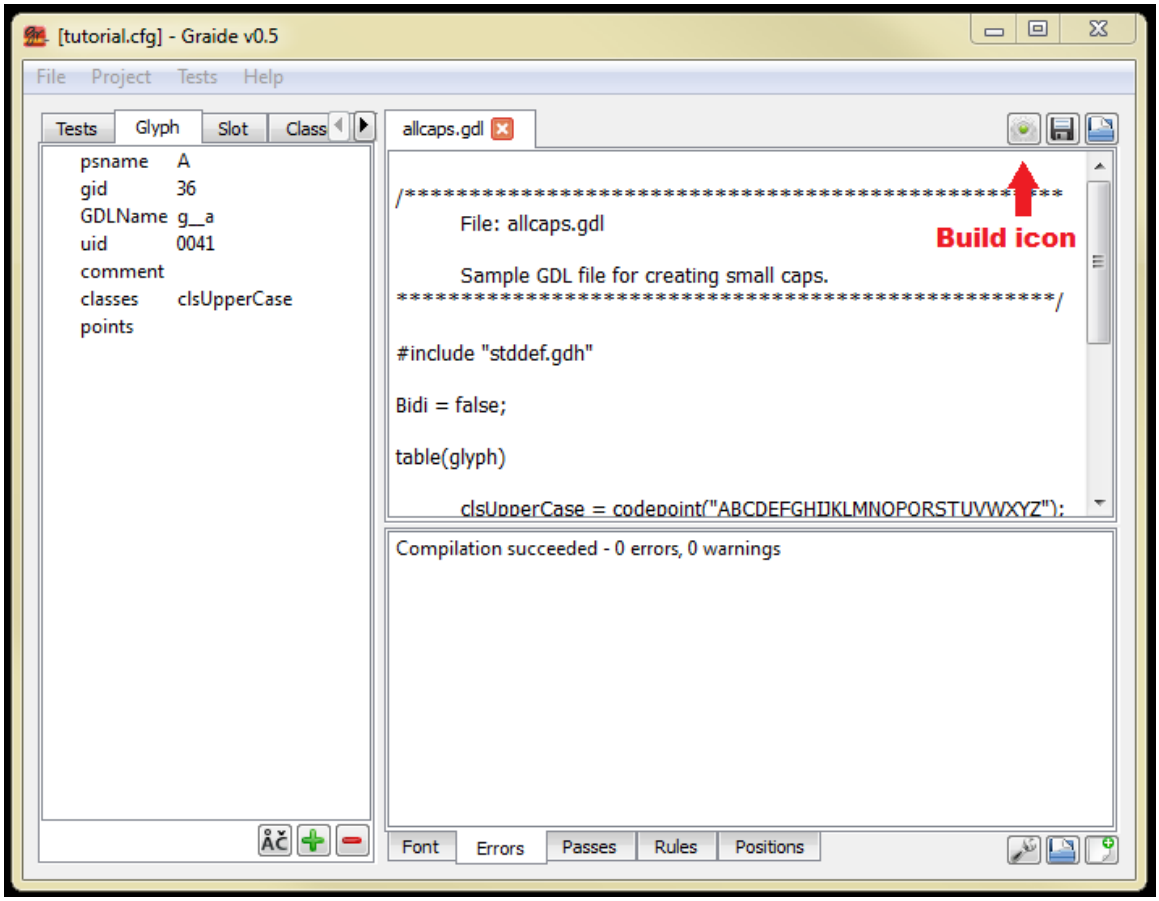
Click OK. The contents of the allcaps.gdl file now appears in the code (upper right-hand) pane. In the lower right-hand pane, the Font tab shows the glyphs in the DoulosGrTut.ttf font.

Step 4. Check your workspace folder. A file has been created called tutorial.cfg.

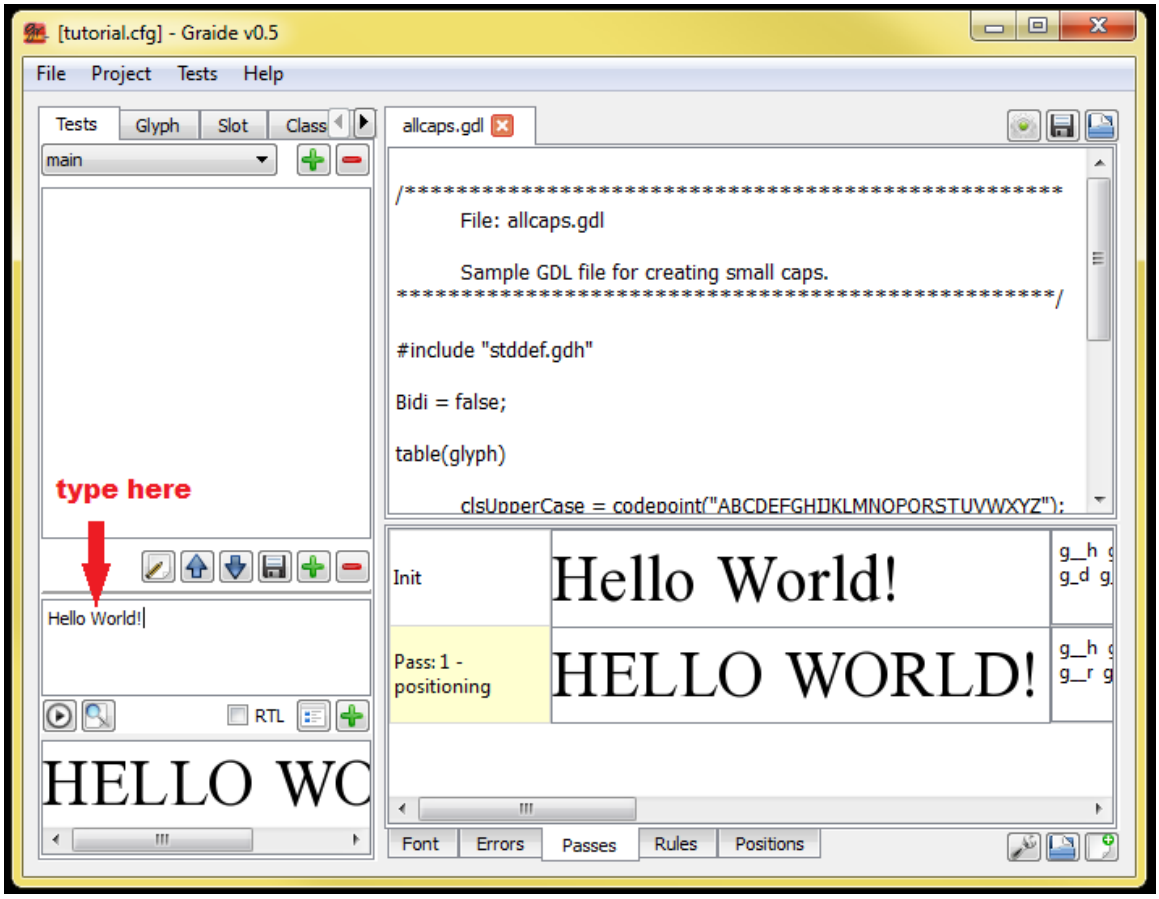
Step 5. Find the uppercase A in the Fonts tab (on the lower right-hand pane) and double-click on it. The left-hand pane switches to the Glyph tab and displays the information for the A glyph.



Step 6. Build the Graphite tables into the font by clicking the gear icon associated with the code pane. In the lower pane, click on the Errors tab to double-check that there were no errors. (Normally the Error tab will be automatically brought into focus if there are errors.) You should see the text “Compilation succeeded – 0 errors, 0 warnings”. Your workspace folder should contain a file called DoulosGrTut.gdx.

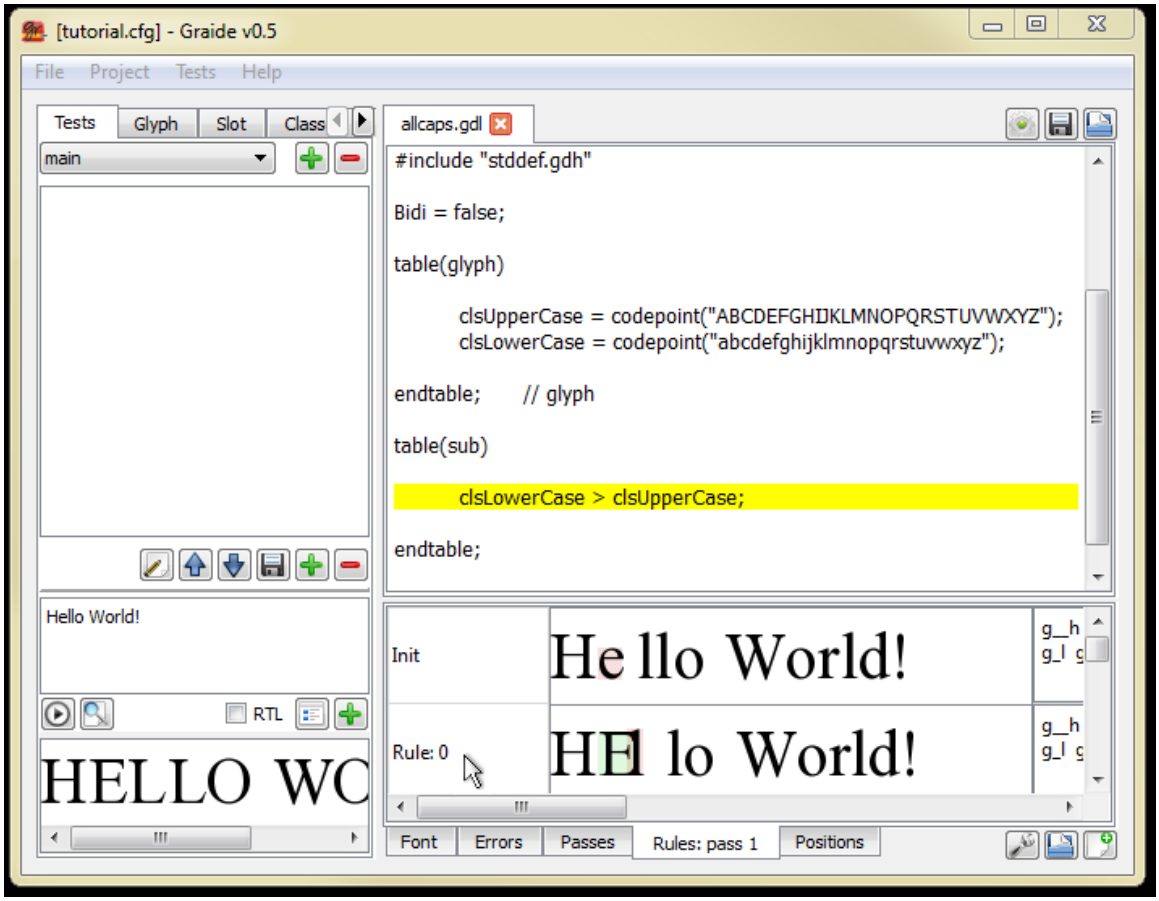


Step 7. Create some test data as follows: Click on the Tests tab again. In the middle (test data) pane, type: Hello world! Click the right-pointing arrow. The bottom (test results) pane shows the results of the Graphite rendering: HELLO WORLD!



Step 8. Graphite will show you which rules were run in the process of rendering. In the case of this font, there is only one rule in the GDL program. If the Passes tab is not selected, click on it. Double click on the box that says “Pass: 1 - substitution”. Then double-click on any box that says “Rule: 0”. Graide will display and highlight the single rule in the code pane:

```
clsLowerCase > clsUpperCase;
```



We will begin to learn about rules in Unit 2.

Exercise 1b

For the second exercise, we will deliberately introduce some errors into the allcaps.gdl program and discover how Graide assists in correcting them.

Step 1. Click on the Classes tab in the left-hand pane. Double-click on the cell labeled "clsUpperCase". Notice how the following line is highlighted:

```
clsUpperCase = codepoint("ABCDEFGHIJKLMNOPQRSTUVWXYZ");
```

Step 2. Change the name of the class to "clsUppercase" and change the letter "L" in the `codepoint` function to "X":

```
clsUppercase = codepoint("ABCDEFGHIJKXMNOPQRSTUVWXYZ");
```

(Note that the line does not need to be highlighted in order to edit it; the highlighting is simply a convenient feature of Graide.)

We have now introduced two bugs into the program: a syntactic error (incorrect class name) and a logical error (mismatch between the upper- and lower-case letters).

Step 3. Build the font by clicking on the gear icon. The Error tab becomes active with an error message indicating the syntax error:

```
allcaps.gdl(23) : error(3139): Undefined class name: clsUpperCase
```

Double-click on error message, and the rule with the (now) incorrect class name is highlighted.

Step 4. Correct either the class definition or the rule so that the names match. Build again by clicking the gear icon. You should see a "Compilation succeeded" message.

Step 5. Click on the Tests tab in the left-hand pane. If your test data has disappeared from the middle pane, retype it: Hello world! Click the right-pointing arrow icon. You will now see the result of Graphite's rendering which includes the bug that was introduced: HEXXO WORXD!

Step 6. If you like, fix the bug in the definition of `clsUppercase`, rebuild and rerun the test to see the correct results.

Unit 2: A very simple GDL program

The most fundamental aspect of GDL is the set of rules that define the script rendering behavior. The following is an example of a rule:

```
glyphid(68)  >  glyphid(69);
```

This simple rule is made up of two parts: the left-hand side (input) and the right-hand side(output). It says to replace every instance of the letter "a" (glyph 68 in the Doulos Graphite Tutorial font) with the letter "b" (glyph 69).

Note that Graphite works only with glyphs. However, it is frequently easier for the programmer to work with Unicode values. So instead, one could write:

```
unicode(0x61)  >  unicode(0x62);
```

Graphite still works with the glyphs that these codepoints represent. The Graphite compiler will use the font's cmap to replace the Unicode Scalar Values with the appropriate glyph IDs in the compiled font.

Notice that hexadecimal numbers (which are the standard way to specify Unicode codepoints) must be prefixed with "0x" within the `unicode` function. You can also use decimal numbers; the following rule is equivalent to the one above:

```
unicode(97) > unicode(98);
```

An equivalent of the `unicode` function is the standard syntax for Unicode codepoints:

```
U+0061 > U+0062;
```

Note that is *not* necessary to include "0x" to indicate that the value is hexadecimal when using this syntax. The "U" must be capitalized.

Another way to reference glyphs is using the `postscript` function, which identifies a glyph using its Postscript name:

```
postscript("a")  >  postscript("b");
```

Note the quote marks surrounding the Postscript name string.

Rules are organized into tables. There are three kinds of tables that contain rules: the *linebreak* table (abbreviated `lb`), the *substitution* table (`sub`), and the *positioning* table (`pos`). The following syntax is used to indicate a table:

```
table(table-name)
      rules in the table
endtable;
```

Because the rule above is substituting one glyph for another, it would be located in the `substitution` table.

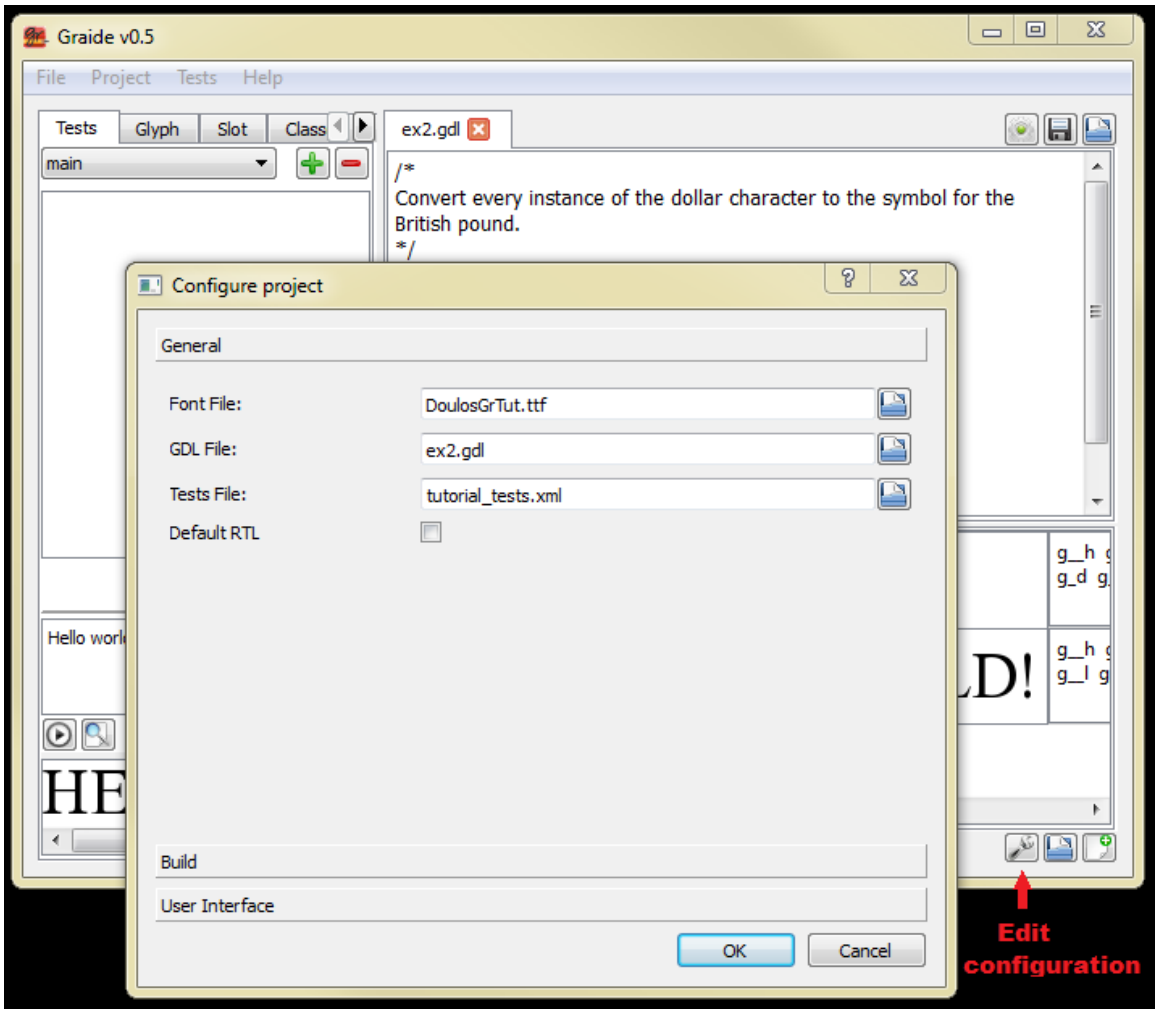
To access the standard definitions and abbreviations, add the following statement at the beginning of your file:

```
#include "stddef.gdh"
```

Exercise 2

Write a simple GDL program to convert every instance of the dollar character to the symbol for the British pound. (Note that you are not trying to convert the actual currency values, just the currency symbol itself!) The Unicode codepoint for the dollar is U+0024 and the codepoint for the British pound is U+00A3.

Use Graide to create and debug your new program. Click on the configuration icon to edit the configuration.



Enter a new GDL file name. You can leave the font name the same (DoulosGrTut.ttf). Click OK and a new tab will open in which for you to enter your code. After writing your GDL, click the gear icon to build the Graphite tables and insert them into the font.

Note that adding the new Graphite tables removes the old behavior implemented in Exercise 1. If you want to preserve the old font, make a separate copy of DoulosGrTut.ttf.

Test your font using some text that includes the dollar sign.

Unit 3: The glyph table

In addition to the tables that contain rules, there are tables containing other information. One of these is the *glyph* table. Among other purposes, the glyph table can be used to define classes of glyphs, and/or give meaningful names to single glyphs. The glyph table is marked by the following syntax:

```
table(glyph)
      glyph class definitions
endtable;
```

The glyph class definitions include lists of glyphs or codepoints that comprise the members of the class. The syntax of the glyph class definition is the following:

```
class-name = list-of-glyphs;
```

For instance, you can give names to a group of digit characters as follows:

```
gZero = U+0030;
gOne = U+0031;
gTwo = U+0032;
gThree = U+0033;
gFour = U+0034;
gFive = U+0035;
gSix = U+0036;
gSeven = U+0037;
gEight = U+0038;
gNine = U+0039;

clsDigit = (U+0030..U+0039) ;
```

Notice the final statement defines a class of glyphs that includes ten digits. It could also have been written as follows:

```
clsDigit = (gZero, gOne, gTwo, gThree, gFour, gFive,
      gSix, gSeven, gEight, gNine);
```

Parentheses are needed around the list of glyphs when more than one item is included. (Note: the use of an initial "g" to indicate single-glyph definitions and "cls" to indicate multiple-glyph classes is simply a convention.)

In Unit 2 we mentioned briefly that `glyphid` can be used to reference a glyph directly, i.e.:

```
glyphid(number);
```

where the number is the ID of the glyph in the font. (Note that `glyphid` can be less useful than `postscript` because it is more font-specific and therefore makes a program that is not as easily converted to work with a different font.)

A fourth way to identify a glyph is using he `codepoint` function. It takes as an argument either a string that contains a list of characters or a numerical value. The codepage can be included as an optional second argument (Codepage 1252 is the default).

```
codepoint("a");
codepoint(97, 1252);
```

As indicated in the example above, the `codepoint`, `unicode`, and `glyphid` functions can specify ranges of values by including the ".." syntax, for instance:

```
clsLowercase = codepoint(97..122);   // a through z
```

(Note that a comment is preceded by two slashes as shown above.) When a glyph has been given a name within the glyph table, that name can be used within a rule in place of the functions. The following is equivalent to the rule discussed in Unit 2 that replaces a's with b's.

```
table(glyph)
      g_a = unicode(0x61);
      g_b = unicode(0x62);
endtable;

table(sub)
      g_a > g_b;
endtable;
```

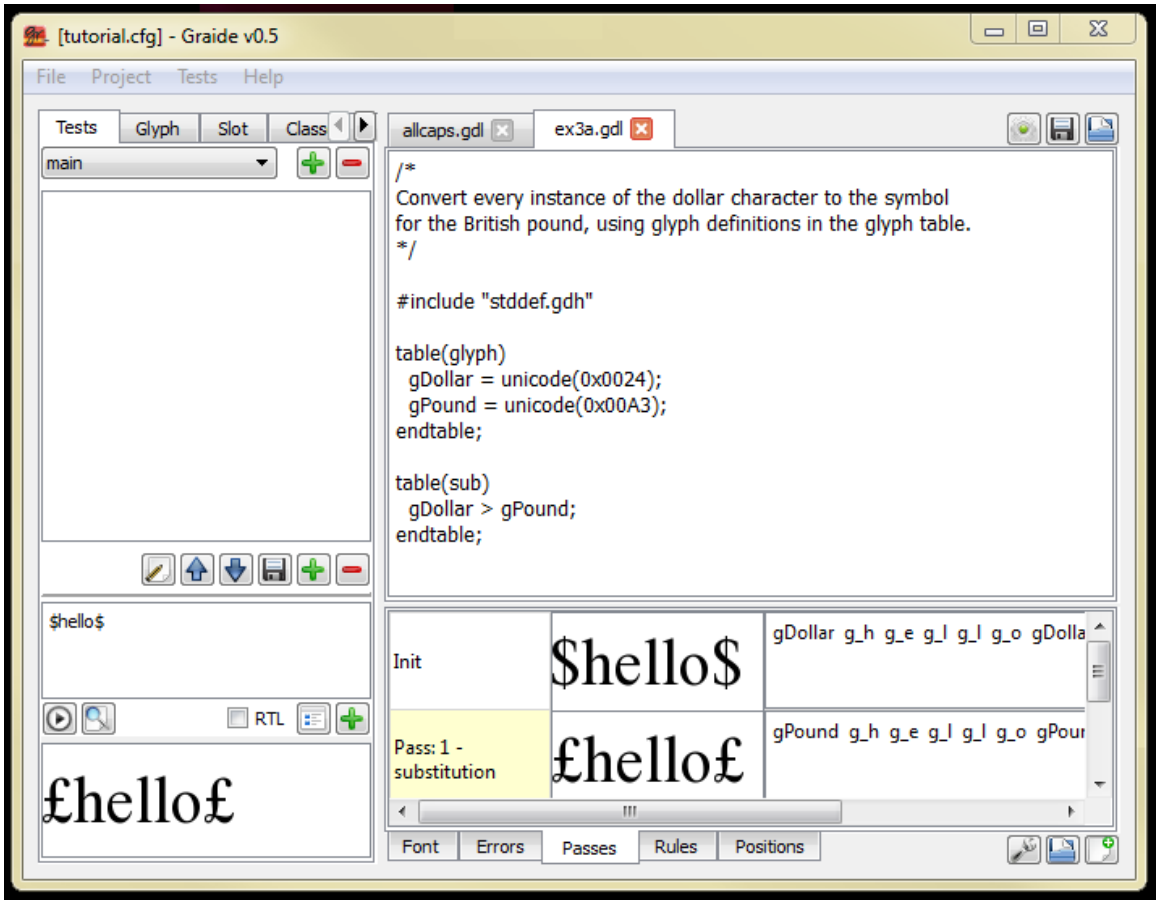
Exercise 3a

Rewrite your program from Exercise 2 to define the dollar and pound glyphs in the glyph table.

Exploring Graide: the Passes and Glyph tab

After Exercise 3a is correct, let's take a few minutes to explore a couple features of Graide. Enter the following data in the text data pane: $hello$. Click the right-pointing arrow to run it. The result should look like: £hello£.

The Passes pane is displayed automatically. It shows the original glyphs (labeled "Init") and the result of Graphite's rendering for each pass. Since there is only one pass in this program, there are two rows in the Passes pane: the original glyphs ("Init") and the output of Pass 1. The second column shows the glyphs that were output for the pass, and the third column shows the corresponding glyph names.

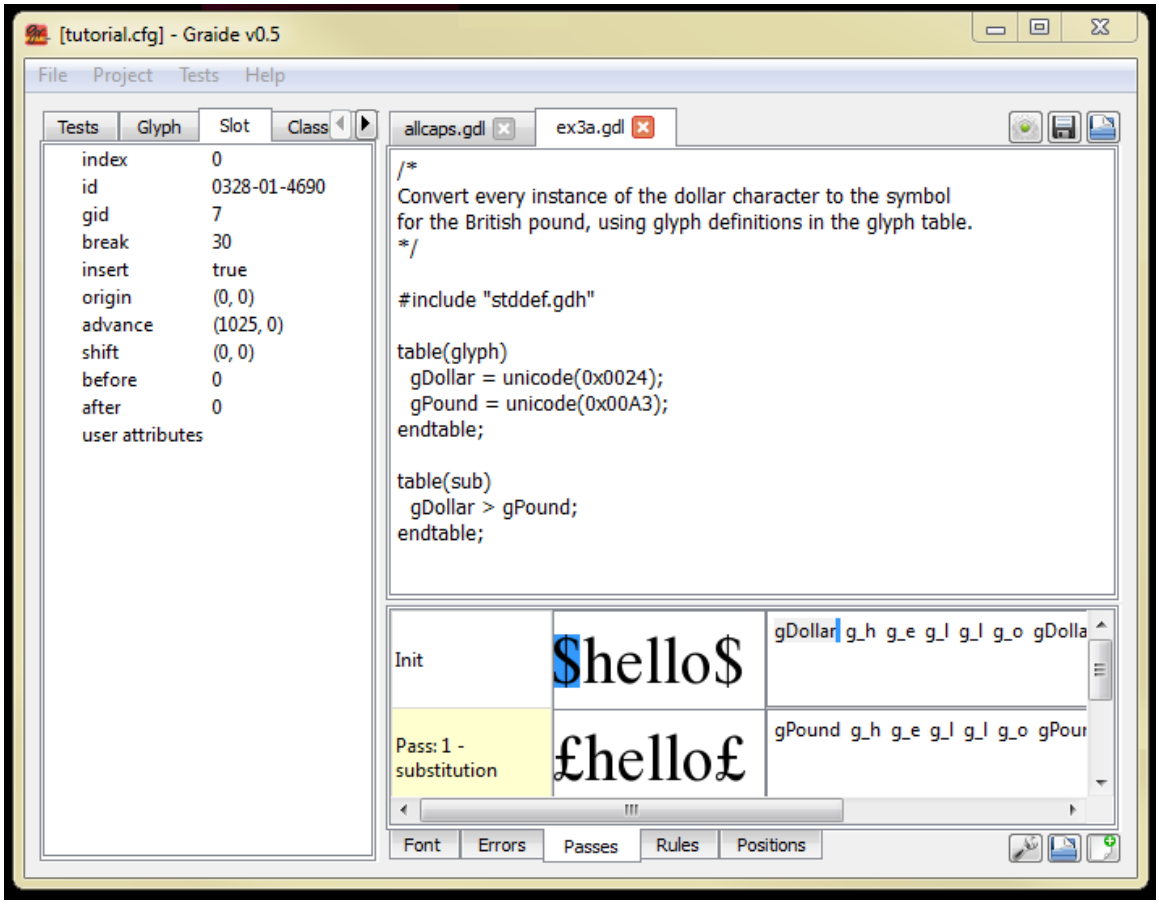


Glyph names are the names given to single glyphs in your GDL program. For instance, if your GDL source code included the line

```
gDollar = unicode(0x0024);
```

you will see “gDollar” twice in the third column of the first row.

Click on one instance of “gDollar”; you’ll notice that the Slot tab is displayed in the left-hand pane. We will discuss the Slot tab later. For now, click on the Glyph tab. This area shows basic information about the dollar (‘$’) glyph from the font and your GDL program.



You can also click on the glyph itself in the second column of the Passes tab to select it. But notice that you must carefully click on *the black part of the glyph;* clicking on white pixels inside or near the glyph will have no effect. (So clicking on the glyph name in the third column may be easier!)

Notice that some of the glyphs in your test data were not given names in your GDL program. For these glyphs Graide generates a name based on the Postscript name from the font. The autogenerated name has "g_" prepended, and any uppercase letters are replaced by an underscore plus the lowercase letter. For instance:

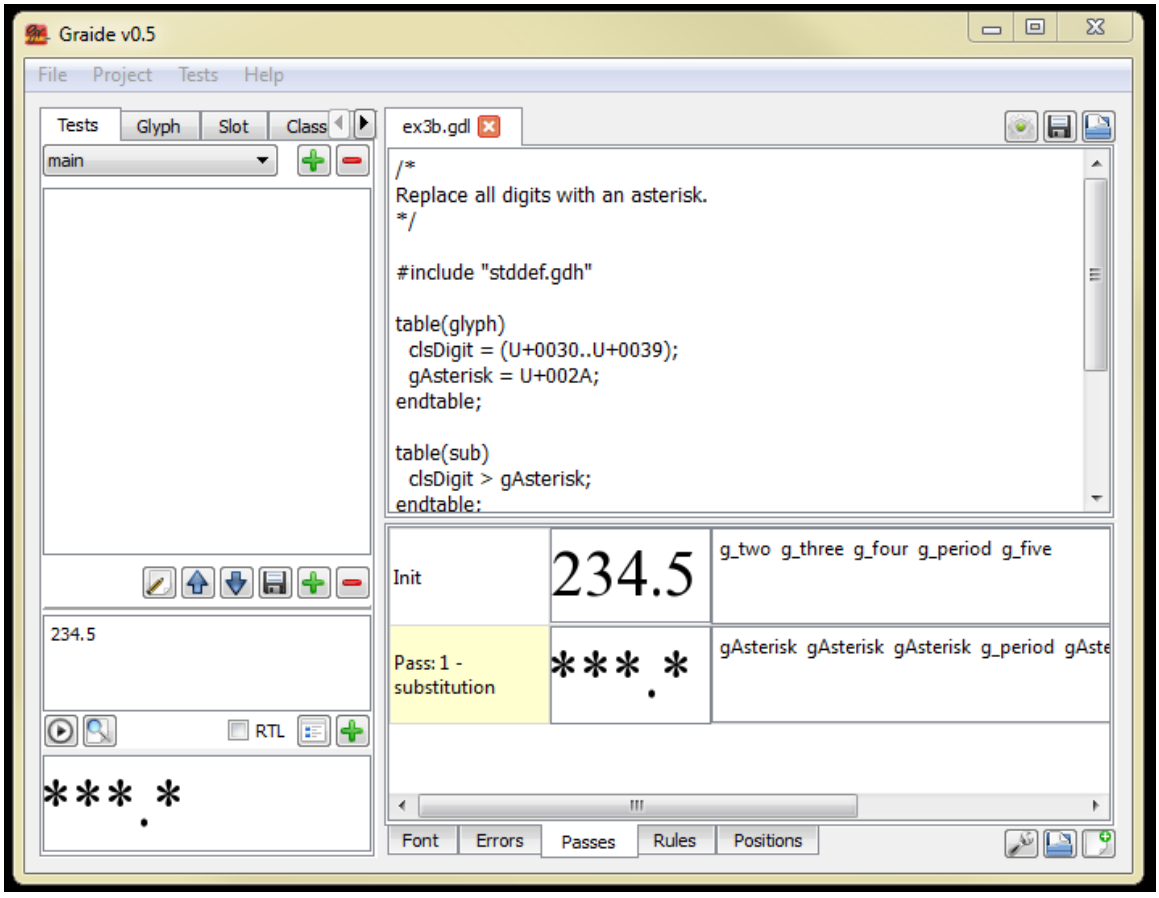
Glyph	**Postscript name**	**Graide name**
a	a	g_a
B	B	g__b
5	five	g_five
(	parenleft	g_parenleft
È	Egrave	g__egrave

Exercise 3b

Write a program to replace all digits with an asterisk. Use the glyph table to define a class containing the digit characters.

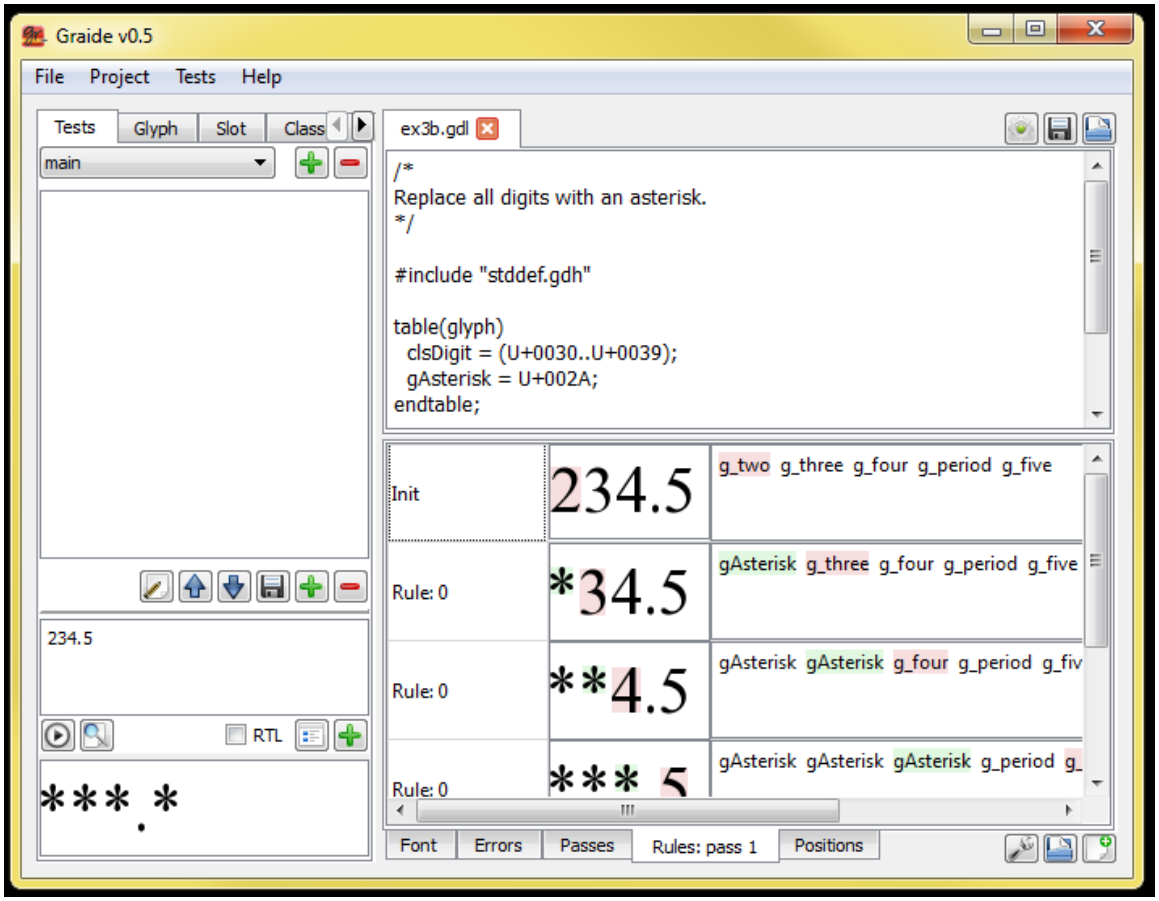
Exploring Graide: the Passes and Rules tabs

After Exercise 3b is correct, let's explore another feature of Graide. Enter the following test data: 234.5; click the arrow to render it. The result should be: ***.*.



The Passes pane is displayed automatically. It shows the initial glyphs and the result of Graphite's rendering for the single pass in this program.

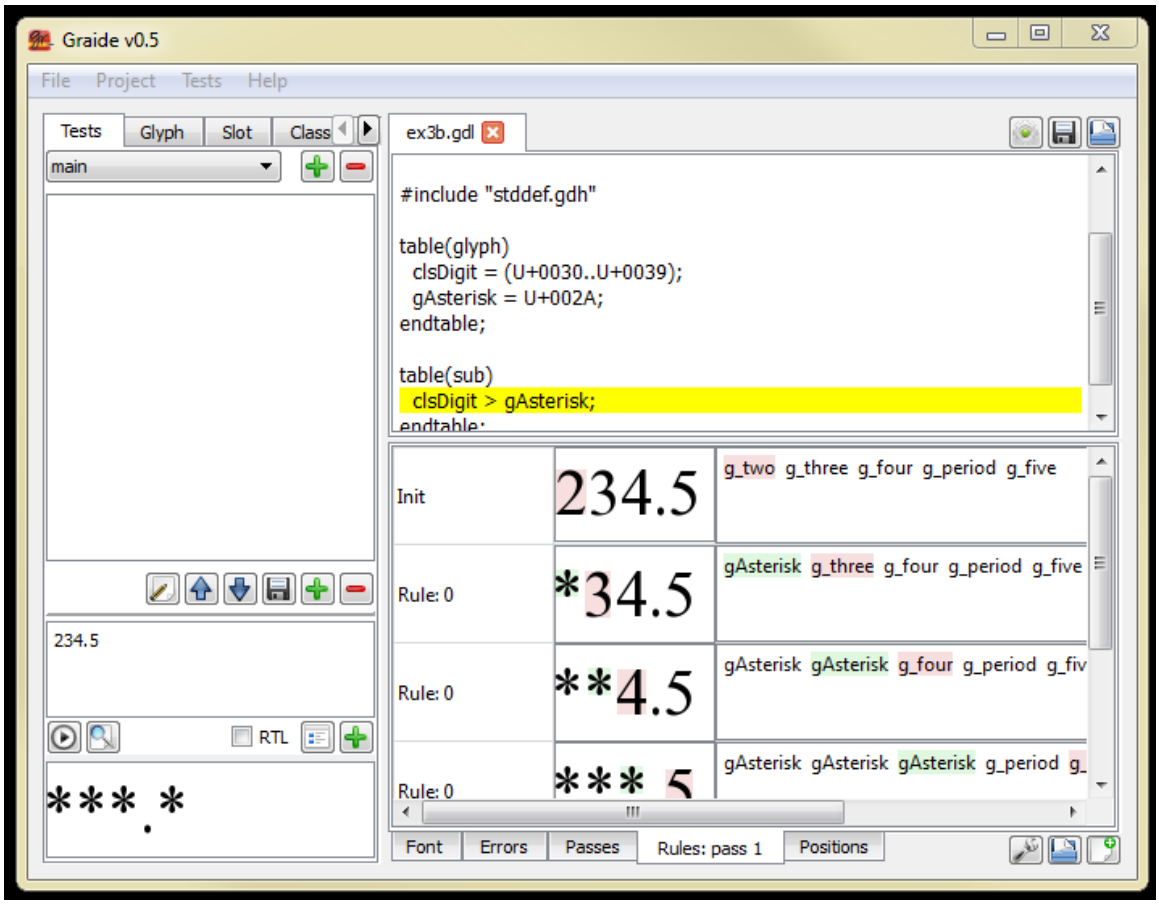
The row labeled "Pass: 1 – substitution" is highlighted in yellow, indicating that there was at least one rule matched in that pass. When you double-click on the yellow cell, the Rules tab appears. Each row of the Rules pane corresponds to one instance of a rule that was matched in pass 1. For this example, if your code looks like the supplied solution, all the rows are labeled "Rule: 0" since that is the only rule in the program.



Note that the output of any rule becomes the input to the next rule fired. So the glyphs shown in row 2 are the output of the first application of the rule, but are also the glyphs that are matched in order to fire the rule the second time.

In each row, the glyphs that are generated—that is, modified—by a rule are highlighted in green. For this program, these are the asterisks. The glyphs that were matched in the previous row are highlighted in pink. As you can see, each row in the Rules tab represents an instance of Rule 0 applied to successive glyphs in the data.

Double-clicking on any of the cells labeled Rule 0 will highlight the rule in the code pane.



Unit 4: Corresponding class items

So far the right-hand (output) side of our GDL rules have contained single glyphs. It is also possible to use a multi-glyph class for substititution.

When one glyph is substituted for another, the Graphite engine determines the position of the input glyph within the input class, and uses the corresponding item from the output class in the substitution. For instance, the following two classes define classes of uppercase and lowercase letters:

```
table(glyph);
      clsUppercase = unicode(65..90);   // A .. Z
      clsLowercase = unicode(97..122);  // a .. z
endtable;

table(sub);
      clsUppercase > clsLowercase;
endtable;
```

When, say, the uppercase “H” is encountered in the input, the corresponding lowercase “h” is placed into the output. Keep in mind that the order of the items in the classes becomes very significant when this mechanism is being used.

Note that you need to use parenthesis around the members of a class when there are more than one. For instance:

```
clsConsVoiced = (gB, gD, gG);
clsConsVoiceless = (gP, gT, gK);
```

Exercise 4a

Write a program to capitalize all text.

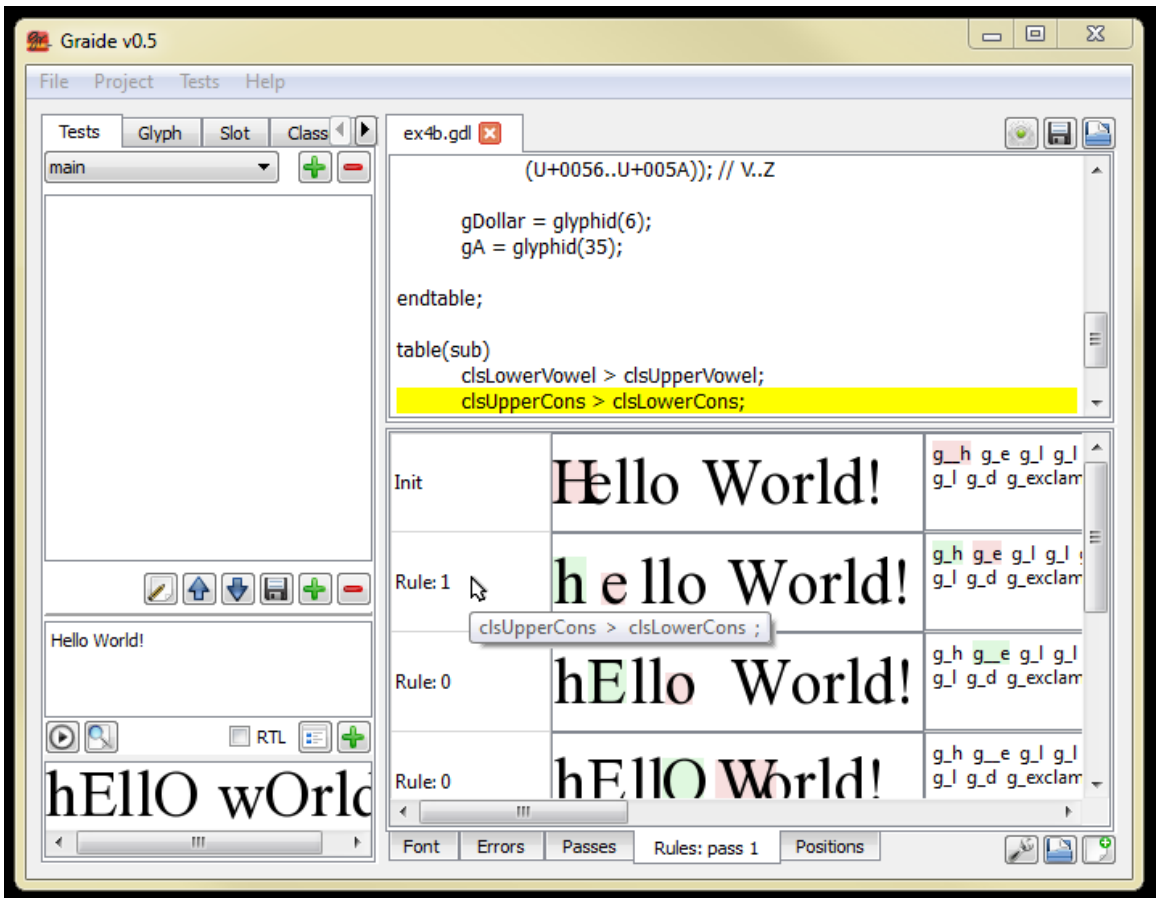
Exercise 4b

Write a program to make all vowels uppercase and all consonants lowercase.

Exploring Graide: the Rules tab with multiple rules

The program you wrote for Exercise 4b should have at least two rules in it. Enter some test data: Hello World! The output should look like: hEllO wOrld! Double-click on Pass 1 in the Passes pane. Notice that both Rule 0 and Rule 1 appear in the Rules pane. Clicking on the box in the left-hand column should highlight the appropriate rule in the code pane.

Also notice that hovering over the left-hand cell containing the rule number will cause the rule to be displayed near the cursor.



Exercise 4c

Write a program to create a Greek transliteration of Roman text, using the mappings in the table below. (You may ignore uppercase for this exercise.) Note that the Unicode

codepoints are not in strict numerical order, and that there are no mappings for the letters j and v.

Roman letter	Greek letter	Glyph	Unicode (hex)
a	alpha	α	03B1
b	beta	β	03B2
c	chi	χ	03C7
d	delta	δ	03B4
e	epsilon	ε	03B5
f	phi	φ	03C6
g	gamma	γ	03B3
h	eta	η	03B7
i	iota	ι	03B9
k	kappa	κ	03BA
l	lambda	λ	03BB
m	mu	μ	03BC

Roman letter	Greek letter	Glyph	Unicode (hex)
n	nu	ν	03BD
o	omicron	ο	03BF
p	pi	π	03C0
q	theta	θ	03B8
r	rho	ρ	03C1
s	sigma	σ	03C3
t	tau	τ	03C4
u	upsilon	υ	03C5
w	omega	ω	03C9
x	xi	ξ	03BE
y	psi	ψ	03C8
z	zeta	ζ	03B6

Compile your program against the Galatia Graphite Tutorial font (`GalatiaGrTut.ttf`) that is provided with your tutorial. It is best to copy the font into your workspace.

Note: If you use lowercase in the hexadecimal numbers representing your Unicode codepoints, you may see a compilation error: "Undefined glyph class: oundingbox". This a bug in the pre-processor, where it interprets the hex digits 'bb' as 'boundingbox'. The work-around is to use 'BB' in instead of 'bb', for instance: `U+03BB`.

Unit 5: Deletion and insertion

Deletion. The most common use for deletion is to replace two glyphs with one. Strictly speaking, this involves one replacement and one deletion. The deleted slot is indicated by an underscore in the right-hand side (output). For instance, the rule below replaces a base character and a diacritic with a single character containing both the base and the diacritic.

```
clsBase clsDiac > clsBasePlusDiac _;
```

Notice that there must always be the same number of items (class names or underscores) on the left- and right-hand sides.

Usually when doing a deletion, you are not, strictly speaking, totally deleting a glyph but rather merging it with another. When this is true you need to specify the slot the deleted glyph is being merged with. This is done by explicitly associating the merged glyph in the output with both of the input slots. This makes it possible for the user to select the single visible glyph and by modifying it, modify both of the underlying characters. For instance, to create this association in the above rule, use the following syntax:

```
clsBase clsDiac  >  clsBasePlusDiac:(1 2) _;
```

This states that the base-plus-diacritic glyph is associated with slots 1 and 2 in the input, i.e., the base character and the diacritic.

Insertion. Insertion is the opposite of deletion. Generally, what occurs is that a single item in the input is replaced by two or more glyphs in the output. In the case of insertion, the underscore occurs in the left-hand side to show the location of the inserted slot. For instance, the rule below is the opposite of the one above:

```
clsBasePlusDiac _  >  clsBase:1 clsDiac:1;
```

As with deletion, in the case of insertion you are rarely inserting a glyph from thin air, but rather splitting the information from one glyph into two (or more). So you can see above that association is made explicit in this situation as well. We indicate that the two resulting slots are both associated with the single input slot. This has the effect that selecting the single underlying character will cause both surface glyphs to be highlighted.

Exercise 5a

Revise your Greek program from Exercise 4c to replace ‘th’ with theta (U+03B8), ‘ph’ with phi (U+03C6), and ‘ps’ with psi (U+03C8). Ensure that selecting the single surface glyph will cause both underlying characters to be selected.

Delete any instances of the letters “j” and “v”. (How will you handle the issue of association?)

Exploring Graide: glyph deletion

Run the following test data: jack in the box. The output should look like: αχκ ιν θε βοξ.

Bring up the Rules tab for pass 1. Notice that in the first row, the ‘j’ is highlighted in pink, but there is no corresponding green glyph in the second row. This is because the ‘j’ has been deleted. Also in the seventh row, both the ‘t’ and ‘h’ glyphs are highlighted in pink, but in the following row, only the theta (θ) is shown green.

Exercise 5b

Write a program to automatically insert a “u” following every “q”. Use associations to ensure that selecting the “q” will automatically select the “u” as well.

Compile it into the DoulosGrTut.ttf font.

Exploring Graide: glyph insertion

The Slot tab shows information about a glyph as it has been modified by a Graphite rule. For instance, it will indicate the association of an inserted glyph. FINISH THIS…….

Unit 6: Context

Often it is desirable to include items in your rule that are not modified by the rule but are simply used for matching. These items are specified in the rule context. The syntax for the rule then becomes:

```
left-hand-side  >  right-hand-side  /  context;
```

Each item from the left- and right-hand sides is represented by an underscore in the context. (Note that this is a somewhat different meaning of an underscore than we saw in the previous unit for insertions and deletions.)

```
gA gB gC  >  gX gY gZ  /  gP _ gQ _ _ gR;
```

The above rule replaces A with X, B with Y, and C with Z. But it only is fired in the situation where gP, gQ, and gR occur as specified in the context, i.e., 'P A Q B C R'. Notice that there is a one-to-one correspondence between the items in the left-hand side, the items in the right-hand side, and the underscores in the context.

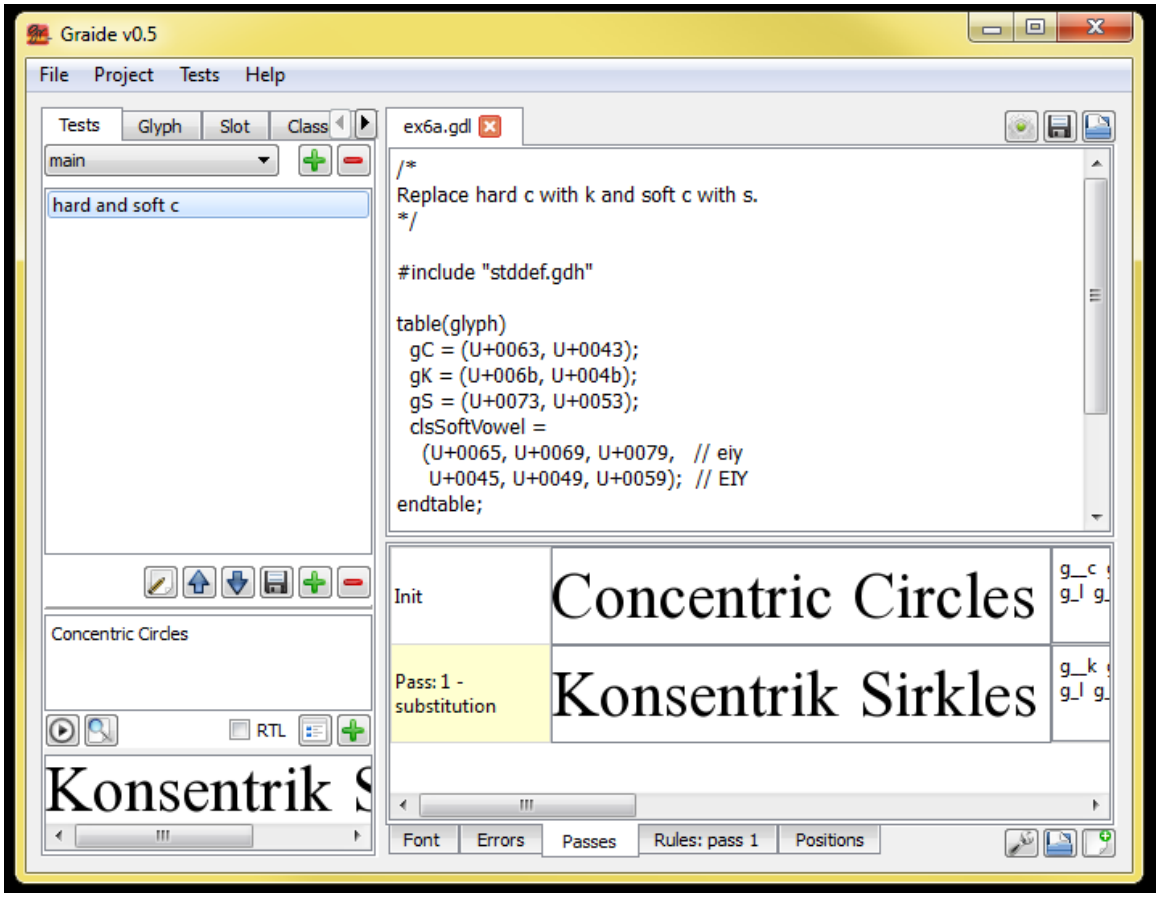
Exercise 6a

Write a program to replace every instance of the letter c. Replace soft c's with 's' and hard c's with 'k'. (Use the rule that soft c's preceed e, i, and y, and hard c's preceed a, o, u, and consonants.) Make sure that the case of the substituted letter matches that of the original "c".

Exploring Graide: Creating a test suite

Graide allows you to store a suite of test data as part of your project. The project configuration includes the name of your data file. Clicking on the Configure project button will enable you to see or change the name of the file that holds the tests. If you set it as suggested in Exercise 1, the file should be called {style:AppText tutorial_tests.xml}. If it is currently empty, set it now.

Choose the Tests tab in the left-hand pane. Click on the green + button to create a new test. Give your test a descriptive name, say "hard and soft c". In the Text field enter: Concentric Circles. This is the data that will be run through Graphite. Click OK. The name of your test should show up in the Tests tab. Double-click the name of the test to run it, or use the Run arrow below. The result should look like: Konsentrik Sirkles.



Exercise 6b

Extend your program from Exercise 5b to always insert a lowercase u unless both the "Q" and the following letter are uppercase, in which case an uppercase "U" is inserted.

Exercise 6c

Extend your Greek transliteration program from Exercise 4c or 5a to replace a sigma with the word-final form ('ς', U+03C2)) where appropriate. Hint: use two rules, the first recognizing the situation where the sigma is *not* word-final.

Unit 7: Glyph attributes

Besides defining classes of glyphs, a further use of the glyph table is to define *glyph attributes*. The values of glyph attributes are constant for every instance of a given glyph (unlike *slot attributes* which are discussed below). There are several pre-defined glyph attributes, and GDL programmers can define their own.

The most common way of defining a glyph attribute is to include it in the glyph class definition, as follows:

```
clsExample  =  (item1, item2, …, itemN) { attr1 = value };
```

The statement above defines the glyph attribute `attr1` for the members of the `clsExample` class. The value of a glyph attribute must be a number or a boolean.

For the exercise below, you will need to be able to test the value of attributes. This is done within the context of the rule. For instance, the rule below will only fire if the value of `attr1` is true for the matched element of clsExample.

```
clsExample  >  clsModified  /  _ {attr1 == true};
```

Note that as in C++, the assignment operator is "=" and the comparison operator is "==". Also the ! operator means "not", and "!=" means "not equal."

Exercise 7

Rewrite the program from Exercise 6a to use a glyph attribute. Assign each letter a value for the glyph attribute "followsHardC". Then test this glyph attribute in the rule to decide whether to change the 'c' to a 'k' or to an 's'.

Does this approach seem natural and elegant, or contrived? ☺

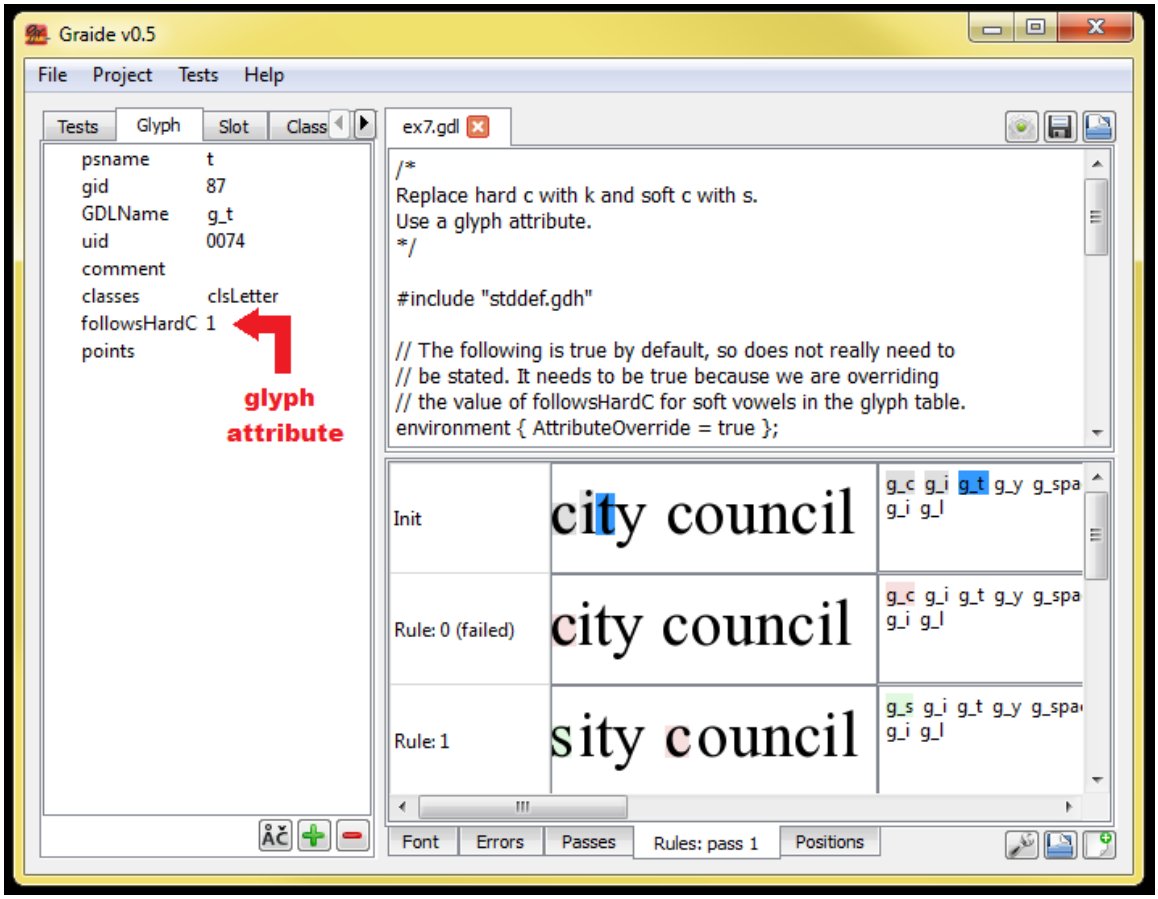
Exploring Graide: failed rules

When you include tests in your rules, some rules may match but not be fired due to the test returning false. These "failed" rules will be displayed in the Rules tab but will be marked "(failed)".

Try running the data: city council. The output should look like: sity kounsil. Double-click on Pass 1 to bring up the Rules tab. You will see several rules that are marked "failed". Notice that there are no green glyphs in these rules' output, since no changes were made. The previous row will show gray glyphs, indicating glyphs that were matched but were left unchanged.

Exploring Graide: glyph attributes

The Glyph tab will show any glyph attributes that are set in your GDL problem. Again, run the string city council and bring up the Rules tab. Select the letter 'i' in the first row (either by clicking the black pixels in the glyph or by clicking 'g_i' in the third column) and choose the Glyph tab. Notice that there is a glyph attribute listed called 'followsHardC' with the value 0 (false). Now select the 't'. The same glyph attribute is listed, but the value is 1 (true).



(If your glyph attributes don't seem to look the same, it could be because your program is somewhat different than the provided solution.)

Unit 8: Slot attributes

Unlike glyph attributes, which are constant for every instance of a given glyph, the values of *slot attributes* may differ for each instance of a glyph. It is the glyph's role as a slot within the context of the entire stream of text that determines the values of its slot attributes.

Slot attributes are set within rules. Most slot attributes are system-defined (and will be discussed further on), but there is also a set of user-definable (that is, programmer-definable) slot attributes, called `user1`, `user2`, etc. More meaningful names can be given to the user-definable attributes using `#define`.

Slot attributes can be tested in the context of the rule in the same way that glyph attributes are.

Intermission

You've got a good introduction to the basic concepts of GDL at this point. Now might be good time to read through the full GDL documentation to get a broader picture before continuing this tutorial. This information is located in the `GDL.pdf` file on the Graphite web site: http://graphite.sil.org.

Unit 9: Multiple passes per table

So far we have been introduced to the substitution table, and have briefly mentioned the existence of other tables containing rules. However it is also possible to have multiple passes per table. These are created by the using the `pass` and `endpass` statements. The first pass should be numbered 1, and the output of each pass becomes input to the following pass.

```
table(sub)
pass(1)
      rules for pass 1
endpass;
pass(2)
      rules for pass 2
endpass;
endtable;
```

Exercise 9

Rewrite your program from Exercises 6a and 7 using slot attributes and two passes. The first pass sets a user-defined slot attribute called "hardC", based on the context in which the c is encountered. Note: there is no need to include the `>` syntax when all you are doing is setting a slot attribute. Use the following syntax:

```
glyph-class { set slot attribute }  /  context;
```

In the second pass, test the value of `hardC` and perform the appropriate substitution.

Have you observed any subtle differences among the results of Exercises 6a, 7, and 9? Does the suggested approach to Exercise 9 introduce a bug? What is the best way to fix it?

Exploring Graide: multiple passes

As you would expect, when you have multiple passes in your GDL program, extra rows appear in the Passes tab. When there are two passes, there are three rows in the pane, one for the initial input, and one containing the output of each pass. When you double-click on a pass, the label on the Rules tab will indicate which pass the rules belong to.

Exploring Graide: slot attributes

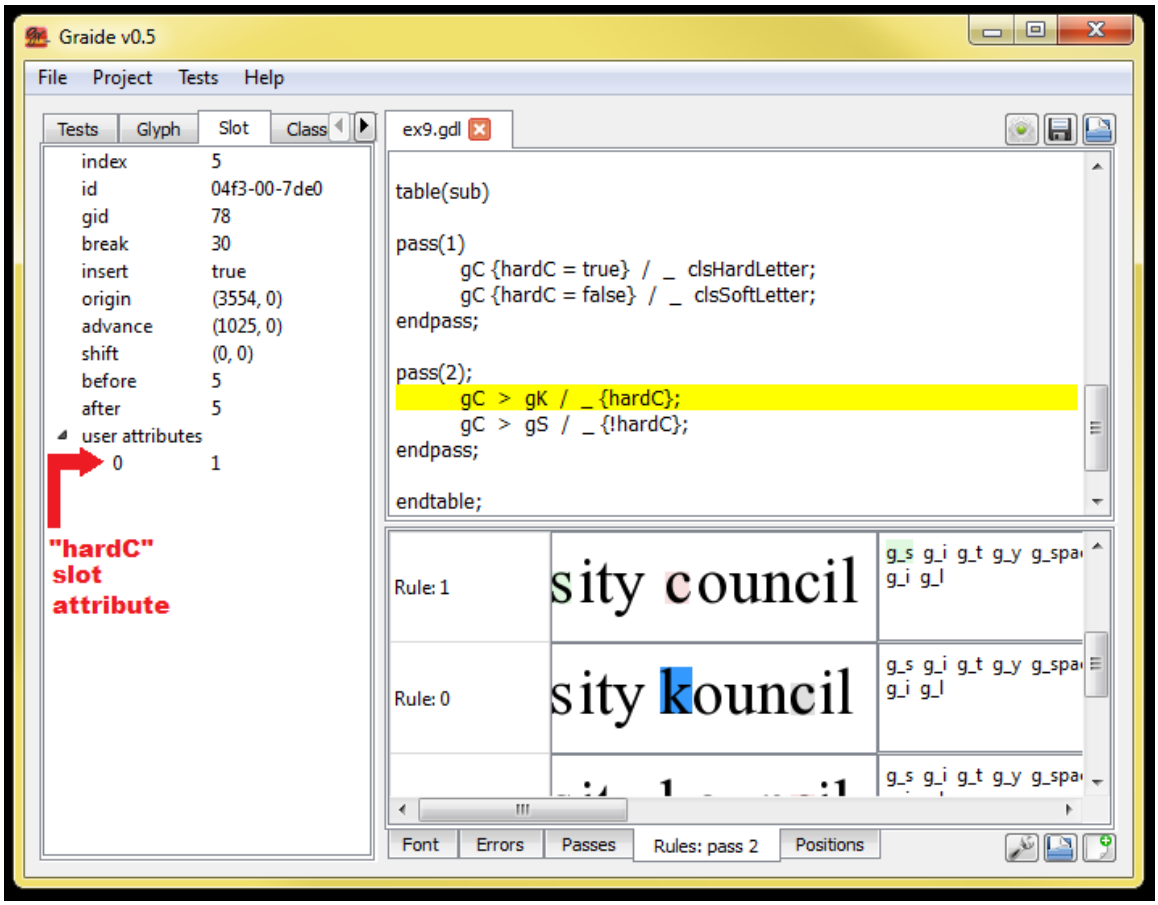
The values of slot attributes can be viewed in the Slot tab on the left-hand pane. Run the following test data: city council. Since Pass 1 is where the slot attributes are set, double-click on Pass 1 to bring up the rules for that pass. As you click on a glyph in the output of a rule, the slot attributes are shown in the Slot pane. The system-defined slot attributes are shown, followed by a list of user attributes.

The actual name of the slot attribute you defined, "hardC", is not shown in Graide. This is because it has been replaced with the generic user-defined slot attribute name by the GDL preprocessor, e.g., "user1". However, a list of user attributes is shown in the Slot tab. If you defined your slot attribute using

```
#define hardC user1
```

your slot attribute is the first in the list. Note, however, that the list in Graide is zero-based, so the first (and probably only user attribute) is labeled 0, not 1.

The first ‘c’ in “council” is an instance of a hard C. You should have run a rule in your GDL program that sets `hardC` to true for this glyph. Click on the green-highlighted glyph that was modified by this rule. The value of user attribute #0 should be 1.



Now click on the pink-highlighted glyph just above. The value of user attribute #0 now shows as 0. This indicates that the rule changed the value of the slot attribute from 0 to 1.

(If you are not seeing these results, it could be because your GDL program is substantially different from the supplied solution. It may be helpful to replace your program with the supplied one in order to experiment with Graide.)

Unit 10: Positioning by shifting

Until now we have discussed only the substitution table, which is used not only for substitution itself but also for insertion and deletion. Now we are going to shift gears and talk about positioning.

Positioning occurs in the *positioning* table, indicated by the following syntax:

```
table(pos)
        rules for positioning
```

```
endtable;
```

The positioning table is always run after the substitution table. Like the substitution table, the positioning table can contain multiple passes. The output of the final pass of the substitution table serves as input to the first pass of the positioning table (except if there is a bidi pass, which comes between them; the bidi pass will be covered in a later unit). If you do not include a positioning table in your program, Graphite will automatically include one positioning pass to position glyphs in the standard way.

There are two ways to make adjustments in position: shifting (including kerning) and attachments. In this unit we discuss the shifting and kerning approach.

Shifting and kerning are performed by means of setting the following slot attributes:

- `shift.x`
- `shift.y`
- `advance.x`
- `advance.y`
- `kern.x`
- `kern.y`

The `shift` attributes adjust the position of the glyph relative to its normal lower-left-hand corner. A positive x value moves the glyph to the right, and a positive y value moves the glyph up. The `advance` attributes adjust the advance width and height of the glyph, which actually have the effect of changing the position of the following glyph. Keep in mind that setting the `shift` attributes alone will have no effect on the position of the following glyph. Kerning—shifting the glyph and adjusting the advance width by an equal amount—can be performed by setting the `kern` attributes, which is actually a short-cut for setting both the `shift` and `advance` attributes.

The values of the `shift`, `advance`, and `kern` attributes are expected to be in units that correspond to the size of the em-square on which the font is based. In your GDL program, you can specify the number of units per em-square using the `MUnits` directive, which can be attached to a `table` or `pass` statement as shown:

```
table(pos) { MUnits = 1000 }
```

The above statement defines a “virtual em-square” of 1000 units high and 1000 units wide. To specify a value in terms of the defined scale, follow a number with an ‘m’.

```
shift.x = 500m
```

The above statement would shift a glyph by half of the width of the font’s em-square. The Graphite compiler scales this number to the actual size of the em-square used in the font, and the engine further scales the resulting number depending on the font size of the text being displayed.

Remember that in the positioning table, because no substitution is occurring, it is not necessary for the rule to include left- and right-hand sides. The syntax is simply:

```
item1 {set attributes}  item2 {set attributes} … / context;
```

and the context part is optional.

Exercise 10a

Write a program to treat all digits as subscripts. Shift them downwards 30% of the size of the font's em-square. (Don't worry about changing the size of the digits.)

Exploring Graide: shift attribute in Slot tab

We've already seen that user-defined slot attributes are shown in the Slot tab. The shift attributes are examples of system-defined slot attributes that are shown there as well.

Run the following test data: H2O. In the Passses tab, click on the '2' or on g_two in the row labeled "Pass: 1 – positioning". Notice there is a slot attribute in the Slot tab called "shift". The value indicates that the glyph has been shifted downward (but not horizontally).

However, you should see that the value is not (0, -300) as you might expect, but rather (0, -614). This is because the slot attributes have been scaled to match the em-square of the font itself, rather than the em-units used in the GDL program. The em-square of the font is 2048, thus -300 has been converted to -614.

Exercise 10b

Write a program to kern the uppercase A when it follows W or V, and vice versa (as in "WAVE"). Experiment to determine the best amount by which to kern.

Exploring Graide: shift attribute in Slot tab

Run the following test data: WAVE. In the Passses tab, click on the 'A' in the row labeled "Pass: 1 – positioning". Notice there is a slot attribute in the Slot tab called "shift" whose value is an (x,y) pair. The x value should be the value you have chosen to kern by, scaled to the em-square of the font.

Unit 11: Glyph metrics

To position glyphs correctly, it is often necessary to make use of the glyph's metrics from within the font. Glyph metrics are numeric values for standard measurements defined for each glyph within the TrueType font. The following metrics can be used in a GDL program:

- `boundingbox.top (bb.top)`
- `boundingbox.bottom (bb.bottom)`
- `boundingbox.left (bb.left)`
- `boundingbox.right (bb.right)`
- `boundingbox.height (bb.ht)`
- `boundingbox.width (bb.width)`
- `advanceheight (ah)`
- `advancewidth (aw)`
- `leftsidebearing (lsb)`
- `rightsidebearing (rsb)`
- `ascent`
- `descent`

Useful abbreviations (as defined in `stddef.gdh`) are shown in parenthesis. Note that `ascent` and `descent` are not metrics of an individual glyph but of the font as a whole.

Exercise 11

Use shifting and/or kerning to position diacritics over lowercase vowels. The diacritics should be attached to the previous base character. Here are some diacritics to include:

- grave: ◌̀ [U+0300] (glyph #250)
- acute: ◌́ [U+0301] (glyph #247)
- circumflex: ◌̂ [U+0302] (glyph #253)
- tilde: ◌̃ [U+0303] (glyph #267)
- macron: ◌̄ [U+0304] (glyph #281)
- diaeresis: ◌̈ [U+0308] (glyph #290)
- hacek: ◌̌ [U+030C] (glyph #264a)

What is required to position them correctly over base characters of widely different widths and heights (eg, m vs. t)?

What is necessary to handle diacritics that have overstriking built in to their glyph metrics?

What is necessary to render i and j correctly with diacritics? (Hint: other glyphs you may find useful include: dotless-i [glyphid 194] and dotless-j [glyphid 195]).

Hint: instead of including numbers directly in the rules, define glyph attributes indicating the amounts by which to shift. You may also need to define a glyph attribute on the base characters as well as the diacritics being positioned. To access glyph attributes for a slot in the rule other than the one being shifted, use the following syntax: WAVE

```
@slot-number.glyph-attr
```

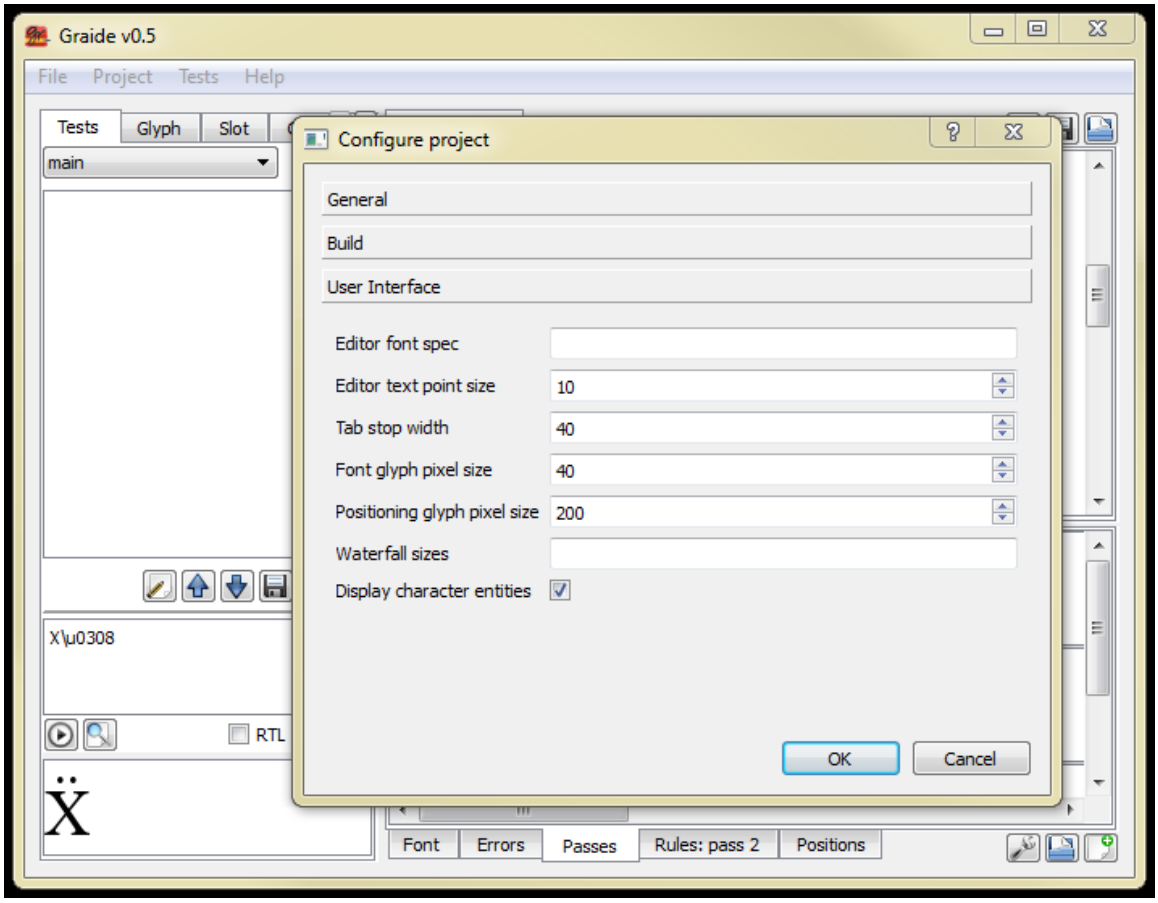
Exploring Graide: non-ASCII test data

If you do not have a keyboard installed that allows you to conveniently type diacritics or other non-ASCII characters, you can enter them into your Graide test data using Unicode codepoints. The syntax is

```
\u<USV>
```

For instance, to enter X + diaeresis, type the following as test data: `X\u0308`. (Do not leave any space after the X.)

If you prefer to always see the Unicode codepoints for non-ASCII data, there is a flag in the Configuration properties dialog to indicate this. Open the section called “User Interface” and check the box labeled “Display character entities.”

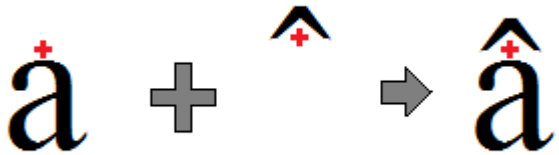


Exploring Graide: Glyph and Slot tabs revisited

Run the following data: m\u0304 I\u0304. Use the Glyph tab to compare the values of the glyph attributes for m and I. Use the Slot tab to compare the amounts by which the diacritic was shifted.

Unit 12: Positioning by attachment

Use of attachment points can be a more elegant way to accomplish positioning than shifting and kerning. With this approach, attachment points are defined on glyphs using glyph attributes. Then there are rules that attach glyphs at their attachment points when they occur within a given context. When two glyphs are attached, they are positioned so that their attachment points exactly coincide.



For instance, the following statement defines an attachment point at the upper center of a base character, where you might want to attach a diacritic:

```
clsBase = glyph-list
      { upperAttPtS = point(bb.width/2, bb.top) };
```

A second statement defines a corresponding attachment point at the bottom of the diacritic:

```
clsDiac = glyph-list
      { upperAttPtM = point(bb.width/2, bb.bottom) };
```

Notice that attachment points generally come in pairs, and so it is helpful to give them corresponding names. A helpful convention that is particularly recommended if you are using Graide is to use an S for the point on the base, meaning "stationary"—i.e., the base is stationary with respect to the diacritic. Similarly, M can be used on the diacritic to indicate that it is "mobile" with respect to the base.

A rule within the positioning table is used to perform the attachment, as shown:

```
table(pos)
   clsBase  clsDiac {attach {to=@1; at=upperAttPtS;
                                    with=upperAttPtM }};
endtable;
```

This rule has the effect of attaching the diacritic (the mobile glyph). The `attach.to` slot attribute indicates the neighboring (stationary) item to which it should be attached, that is, the base character. The `attach.at` attribute names the glyph attribute which is the relevant attachment point on the base character, and `attach.with` indicates the corresponding attachment point on the diacritic. (Remember that `{attach {to=@1; …}}` is equivalent to `{attach.to = @1; … }`.)

Exercise 12a

Rewrite your program from Exercise 11 to position the diacritics using attachment points. Use the Graide visual editor (described below) to specify the attachment points.

Exploring Graide: using makegdl and the attachment point editor

Graide includes a visual attachment point editor. It allows you to see glyphs to be attached on the screen and move them around to achieve the best positions for the attachment points.

Graide creates an XML file to store the point values. (This kind of file could also be generated by other software such as FontLab.) This file is then used to autogenerate a GDL file containing the attachment point glyph attributes. In fact, the process that generates the GDL, called *makegdl*, will generate other GDL source code, including glyph definitions for all the glyphs in your font, and classes based on Postscript names.

To use this mechanism, you must do the following.

Step 1. Set up your configuration appropriately. Go into the configuration dialog and choose the Build section. Enter a name for your attachment point file, eg, ex12a_ap.xml.

Notice that the name of an autogenerated GDL file is supplied for you, based on the name of the font. Click OK.

Start with a new GDL file, or remove any existing positioning rules. Click the gear icon to build the project. (If your GDL file is empty, there will be one warning in the Errors pane.) A file called DoulosGrTut.gdl has been created; open that file in your code pane. You'll notice there are definitions provided for all the glyphs in the font, based on the Postscript names. Toward the bottom of the file are some classes. There is a section called "Point Classes" which is currently empty, but eventually will contain groups of classes that have certain attachment points defined.

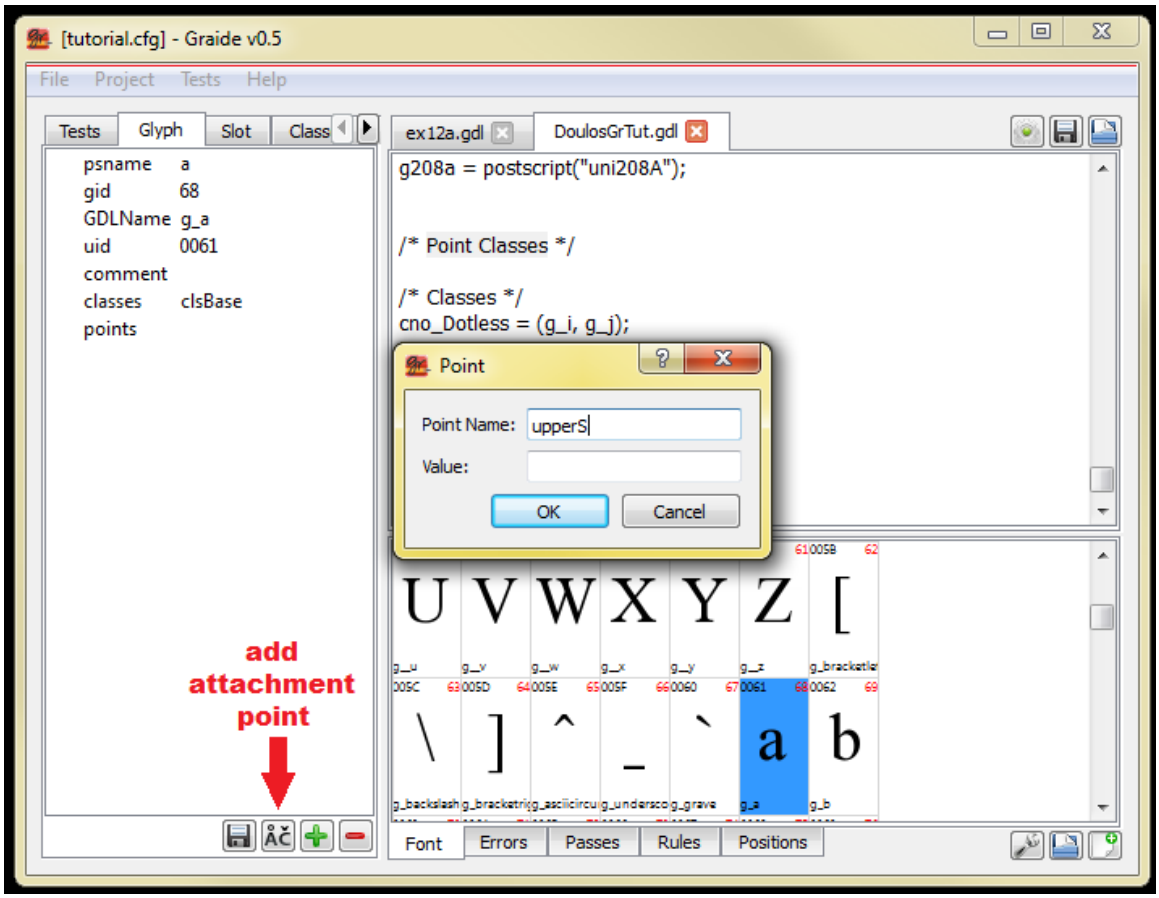
The very last line of the file looks something like this:

```
#include "C:\SomePath\MyWorkspace\mytutorialfile.gdl"
```

When Graide is set up to use makegdl, instead of compiling your GDL file directly, the autogenerated file is compiled which in turn includes your GDL file.

Another file that has been created is the ex12a_ap.xml file; open it in Graide. You'll see a sequence of glyphs with various information, but no attachment points—yet.

Step 2. Define some attachment points. Click on the Font tab and find the lowercase 'a'. Double click on it, and the Glyph tab will appear displaying the glyph. Click on the "add attachment point" button at the bottom of the pane.



We are going to define an upper attachment point. So give the new point the name "upperS". ('S' stands for "stationary", meaning that the base glyph is stationary with respect to the diacritic.) You can enter an initial value of "(0,0)", or leave the Value field blank, in which case it will default to zeros. Click OK.

Build the font again. Search in the DoulosGrTut.gdl file for "upperS". You should find it now defined as a glyph attribute of "g_a".

Return to the Font tab and find the grave accent. (U+0300, glyph #250). Double click on it. Add an attachment point to it in the same way, giving it the name "upperM". ('M' indicates that the diacritic is "mobile" with respect to the base character.) If you rebuild the font and search for "upperM", you should find it defined on the combining grave, "g_gravecomb".

Note: while the Graphite engine and compiler do not make any assumptions about the names of attachment points, Graide will work best if you use pairs of attributes with the forms "xxxS" and "xxxM".

Now let's look more closely at the classes in DoulosGrTut.gdl. If you look in the section labeled "Point Classes" you will see a number of classes present, based on the two attachment points we defined. Notice that there are two classes called "cupperDia" and "cTakesupperDia" that contain the grave and the 'a', respectively. These can be used for writing a rule to perform the upper-diacritic attachment.

There are also classes called "cnupperDia" and "cnTakesupperDia". These contain glyphs that do *not* handle the upper attachment point. These can be used for more complex attachment rules.

Step 3. Write the attachment rule in the positioning table. Your program should look something like:

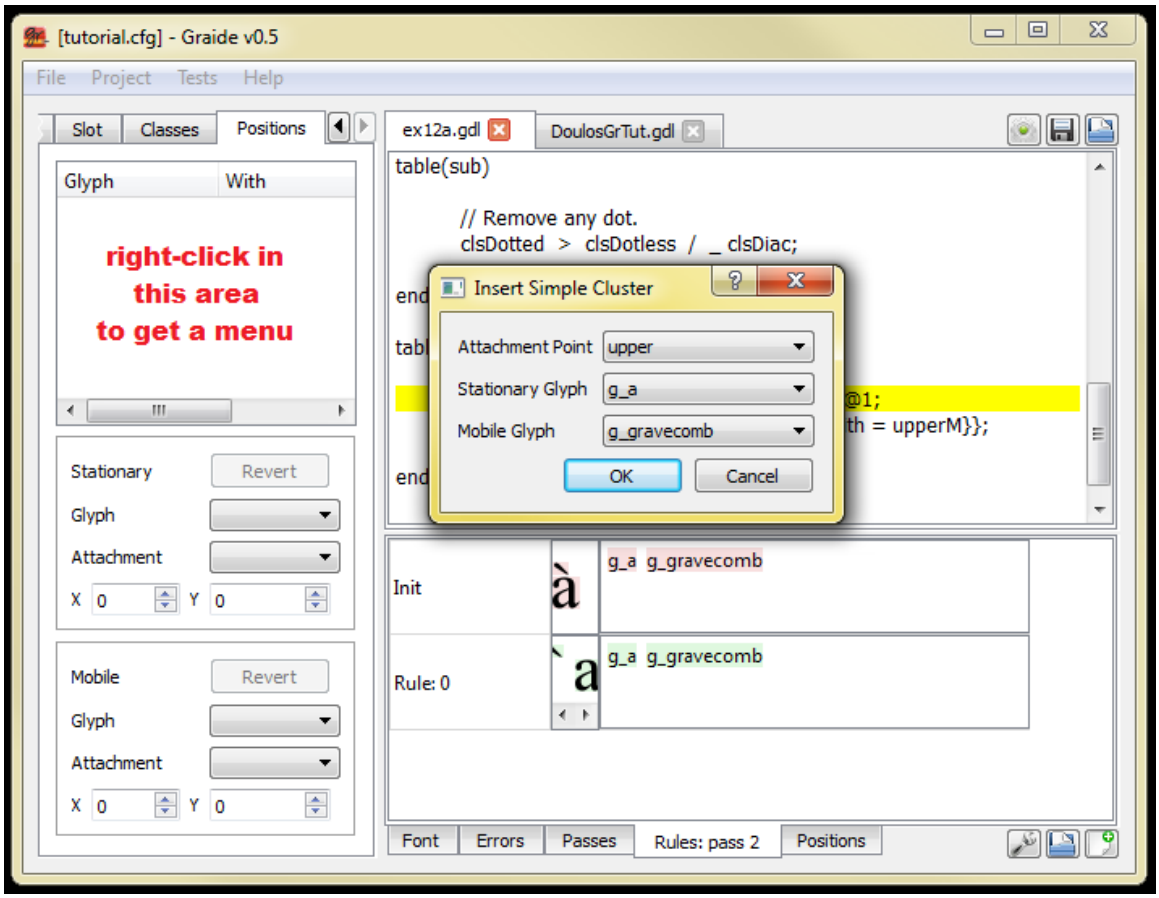
```
#include "stddef.gdh"
table(pos)
      cTakesupperDia  cupperDia {attach {to = @1;
                                  at = upperS; with = upperM}};
endtable;
```

Build your program and run the following test data: a\u0300. By looking in the Rules tab, you should see that the attachment rule is fired, but the results probably don't look very good. This is because the attachment points need to be adjusted. That's the next step.

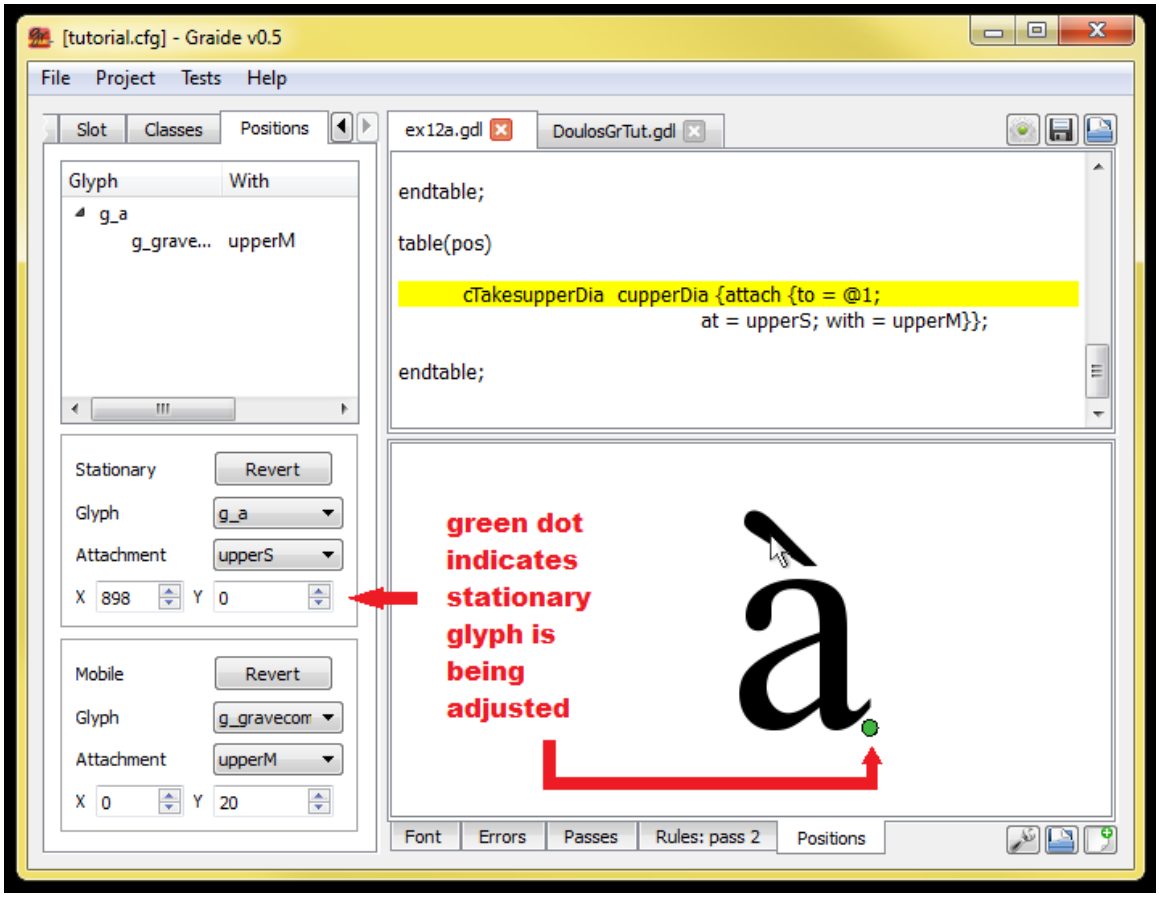
Step 4. Set attachment points visually. Click on the Positions tab in the left-hand pane. What you are going to do is to create a tree-like structure of bases and diacritics that attach to them. The "root" of the tree represents the base(s), and the leaf is the diacritic(s). Right-click in the empty space in the pane and choose "Insert child…". Choose "g_a" which is the lowercase 'a'. The name "g_a" will appear in the list. Then right-click on "g_a" and choose "Insert child…" again. Chose g_gravecomb. Notice that the names of the attachment points that were define are filled in, because these are the only ones available. Click OK.

Step 4. Set attachment points visually. Click on the Positions tab in the left-hand pane. What we are going to do is to create a tree-like structure the represents a cluster of bases

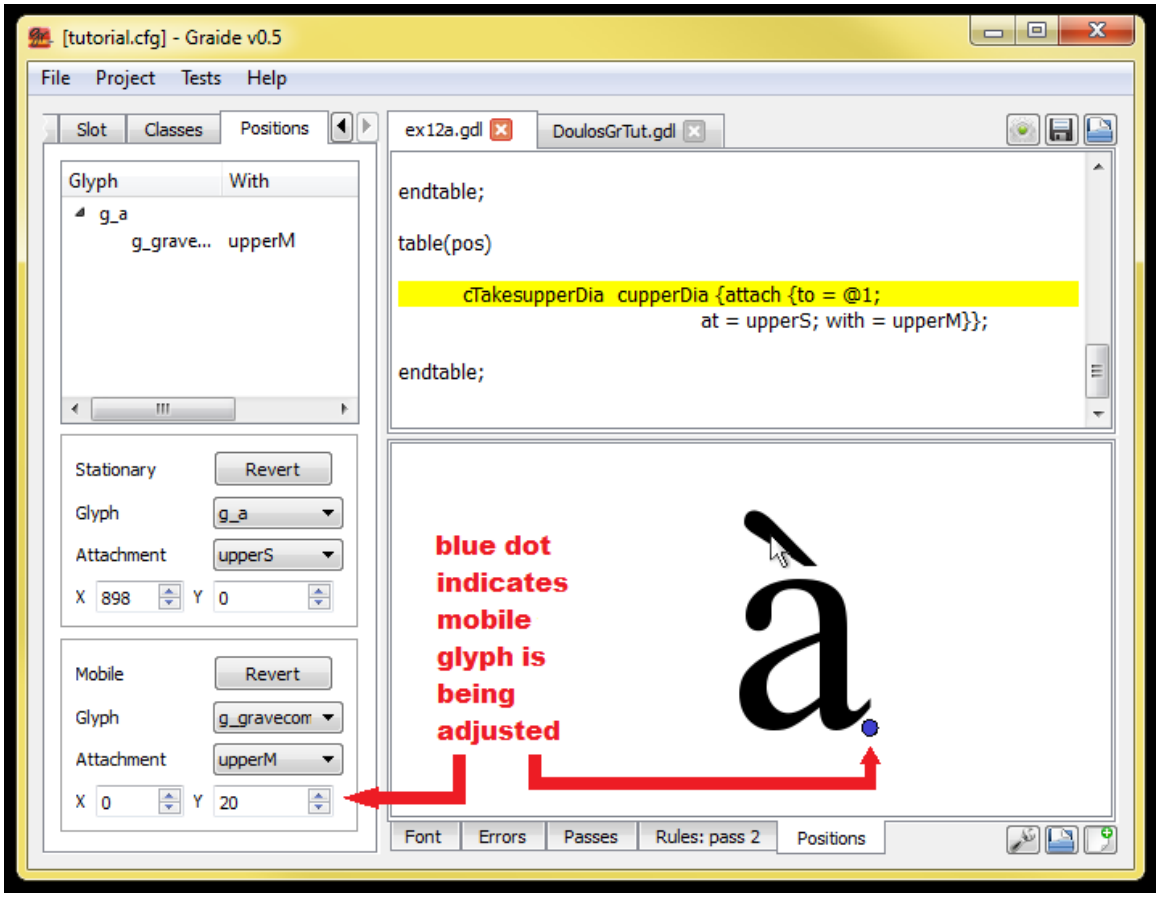
and diacritics that attach to them. The root of the tree represents the base, and the leaves are the diacritics. Right-click in the empty space in the pane and choose “Insert Simple Cluster…”. Choose the “upper” attachment point. The other controls are filled in with the glyphs that define this pair of points (only one each so far). Click OK.



Notice that a tree structure has appeared, and the Positions tab in the lower pane now displays the two glyphs, ‘a’ and the grave combining mark. Drag the grave to an appropriate position on top of the a glyph. As you hold down the mouse, a green dot appears—the green dot means that you are adjusting the attachment point on the *stationary* glyph—upperS. The X/Y coordinates of the point are shown in the left-hand column; notice that they change as you drag the grave.



Another way to adjust the positioning is to change the coordinates of the grave itself—the upperM attachment point. Hold down the SHIFT key and drag the grave again. A blue dot appears to indicate it is the *mobile* glyph that is being adjusted. Notice that the X/Y coordinates of the grave are changing, but the coordinates of the ‘a’ are unchanged.



You may be wondering about the choice between adjusting the stationary glyph as opposed to the mobile glyph. Keep in mind that when you adjust the coordinates of the base glyph, you are adjusting how *all* diacritics that use the set of "upper" attachment points will attach to that base. Similarly, when you adjust the coordinates of the diacritic, you are adjusting how that glyph will attach to all bases. So it is important to carefully consider which kind of adjustment you want to make.

Rebuild the font and rerun your test data. The end result should look a lot like the glyphs you positioned by hand!

Step 5. Add attachment points to two more glyphs. Using the Glyph tab, try adding an upperS attachment point to the 'e' glyph and an upperM attachment point to the circumflex (U+0302, glyph #253). When you return to the Positions tab, the Glyph controls at the bottom of the left-hand pane now offer "g_e" and "g0302". Adjust the positions of the two new glyphs. Notice that you can use the spin controls in the lower-left-hand dialogs as well as dragging the glyphs.

Experiment with the effect of adjusting the stationary points as opposed to the mobile points.

Step 6. Add attachment points to the remaining glyphs. Notice that you can hover over the Glyph dropdown controls and use the scroll wheel on the mouse to quickly cycle through all the glyphs of interest.

Hint: it may be helpful to set the attachment points on the stationary glyphs first, using a symmetrical glyph such as the circumflex, and then set the mobile points using a symmetrical base such as ‘i’.

Define the “upperS” points for dotless-i and dotless-j, and include the substitution rule that removes the dot. You can use the cupperDia class to test for the presence of an upper diacritic.

Exercise 12b

Write a program to position the following lower diacritics using attachment points. (Extend your program from Exercise 12a.)

- grave below: ◌̗ [U+0316] (glyph #249)
- acute below: ◌̖ [U+0317] (glyph #246)
- plus sign below: ◌̟ [U+031F] (glyph #310)
- diaeresis below: ◌̤ [U+0324] (glyph #289)
- ring below: ◌̥ [U+0325] (glyph #298)
- vertical line below: ◌̩ [U+0329] (glyph #306)
- tilde below: ◌̰ [U+0330] (glyph #265)

Exploring Graide: reinitializing the visual attachment point editor

If you already have a cluster present in the Positions tab, right-click and choose “Delete All” to clear those glyphs. Then choose “Insert Simple Cluster” and choose the new lower attachment point.

Exercise 12c

Write a program to position the Greek lower iota diacritic (“ypogegrammeni,” U+037A) correctly under the Greek vowels alpha (U+03B1), eta (U+03B7), and omega (U+03C9). Note that when attached to the eta, the diacritic is not centered but is positioned under the left-hand stroke, as shown:

Compile your program against the “Galatia Graphite Tutorial” font (GalatiaGrTut.ttf).

Extend your program to handle a possible intervening upper diacritic (eg, smooth breathing mark, U+0313 or rough breathing mark, U+0314) between the base character and the lower diacritic.

Unit 13: Features

Graphite includes a mechanism that allows you to create variations of your writing system. This is accomplished by defining *features*. The calling application, along with the

text to render, sends an indication of the feature settings for the range of text, and rules can be fired conditionally depending on the settings of the features.

Features are defined in the feature table. Each feature has an ID and a list of possible settings. The ID is a string that is four characters or shorter. (Originally IDs were numbers, but that approach is deprecated.) You can also include natural-language descriptor strings for use in an application's user interface.

```
table(feature)
      myFeat {
            id = "fea1";
            name.LG_USENG = string("My Feature");
            default = some;
            settings {
                  all { value = 4; name.LG_USENG=string("All")}
                  many{ value = 3; name.LG_USENG=string("Many")}
                  some{ value = 2; name.LB_USENG=string("Some")}
                  few { value = 1; name.LG_USENG=string("Few")}
                  none{ value = 0; name.LG_USENG=string("None")}
            }
      }
      myBooleanFeat {
            id = "fea2";
            name.LG_USENG = string("My Boolean Feature");
      }
endtable;
```

When no settings are defined for a feature, it is considered to be boolean with two settings, true (1) and false (0). The default value for features is assumed to be zero unless otherwise specified.

Rules can be fired conditionally based on the values of features. You can use the `if`, `else`, `elseif`, and `endif` statements to test the values of features.

```
if (myFeat == all)  // parentheses are required
      rules to fire in the case that myFeat has the value "all"
elseif (myFeat >= few)
      rules to handle many, some, and few cases
else
      rules to handle none case
endif;
```

This `if` mechanism tests that every slot in the rule "passes" the test. To test a single slot in the rule but not the others, you can test for the feature in the rule's context, as we saw for testing the values of slot and glyph attributes:

```
gA  gB  gC  >  gX  gY  gZ / _  _ {myFeat >= few}  _;
```

In the above rule, only slot #2 is tested for having the given feature value.

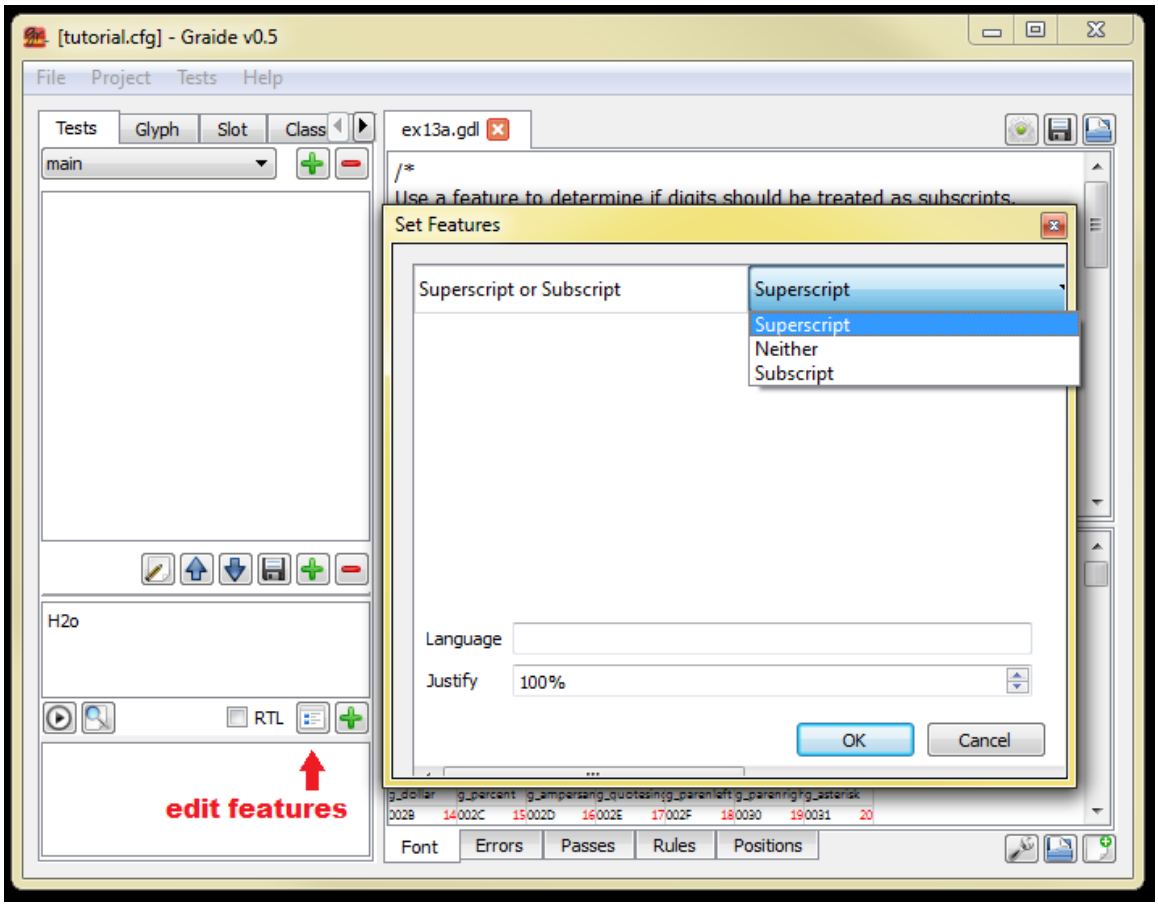
Note that it is the GDL feature name that is used in the test, not the ID (eg, we use `myFeat`, not `fea1`).

Exercise 13a

Extend your program from Exercise 10a. Use a feature with three possible settings to determine if numbers should be treated as subscripts (lowered), treated a superscripts (raised), or left unchanged.

Exploring Graide: setting features on test data

You can use Graide's control to set features on test data.



Note that the feature value(s) you set this way will apply to the entire test string.

Exercise 13b

Extend your program from Exercise 10b to only perform the kerning when the "kern" feature is turned on.

Exercise 13c

Extend your program from Exercise 4c, 5a, or 6b to either display Roman or Greek text depending on the setting of a feature.

Unit 14: Ligatures

A special case of deletion involves the creation of ligatures, where two or more items are replaced by a single glyph representing both or all of them. In the case of a ligature, there are visual components of the final glyph that correspond to each of the underlying characters. There are two parts to this process: first, defining the visual components of the ligature, and second, associating the components with the underlying characters. The first part is done in the glyph table, the second in the substitution table.

Note:Manipulating the visual components of ligatures is not supported in Graphite2, the engine used by Graide, LibreOffice and Firefox. This behavior can only be tested using WorldPad or FieldWorks.

Let's suppose we want to replace the sequence 'ae' with an æ ligature. The left half of the glyph corresponds to the underlying a and the right half to the underlying e. First we define the visual components, which is done using a specially constructed set of glyph attributes in the glyph table. Each component is defined using a statement with the following syntax:

```
component.name = box(left, bottom, right, top)
```

Note that each component can consist of one rectangular area of the glyph. A useful way to define components to to use glyph metrics, especially the bounding box of the glyph. For our æ ligature, the complete glyph definition would look like the following:

```
gAElig = U+00E6 {
   component.a = box(bb.left, bb.bottom, bb.width/2, bb.top),
   component.e = box(bb.width/2, bb.bottom, bb.right, bb.top) };
```

These statements specify that the left half of the glyph is the "a" component, and the right half is the "e" component. Note that use of the glyph metrics causes each component to be half as wide as the full glyph. Also note that in the syntax `component.a`, "component" is a system-defined term used specifically for ligatures, and "a" is the component name chosen by the GDL programmer.

The second part of the process is to associate the character in the input with the appropriate ligature component. This is done in a substitution rule, using slot attributes with the following syntax:

```
component.name.ref = @slot-number
```

So a complete rule to create the æ ligature would be as follows:

```
gA  gE  >  gAElig:(1 2) {component {a.ref = @1; e.ref = @2}}  _;
```

Notice that in addition to mapping the "a" and "e" components to the input slots, the associations must be set as in the case of a normal deletion.

An alternate syntax is:

```
gA  gE  >
   gAElig:(1 2) {component.a.ref = @1; component.e.ref = @2}  _;
```

Exercise 14a

Write a program to create ligatures from the sequences 'oe', 'fi', 'fl'.

Hint: which element of the ligature can be used to uniquely identify the ligature?

Extend your program to create the ligatures only if a feature is turned on.

Exercise 14b

Write a program to substitute fraction glyphs for the sequences '1/2' (U+00BD), '1/4' (U+00BC), and '3/4' (U+00BE). Keeping in mind that ligature components can overlap visually, how will you define the components so that they can be selected and edited? If you choose to allow your components to overlap, what kind of behavior do you observe with the selection mechanism?

Hint: you will probably need to use more than one rule due to the fact that neither the numerator nor the denominator identifies the fraction uniquely. Can you still define the component boxes with a single set of statements?

Unit 15: Bidirectionality

Some writing systems are bidirectional, meaning that they may consist of ranges of text that are written from right-to-left and other ranges that are written from left-to-right. The Unicode Standard contains an algorithm for rendering bidirectional scripts, as well as mixing scripts of different directions.

This algorithm is built into Graphite. Between the substitution and positioning tables there is a special pass called the "bidi" pass. It takes the output of the final substitution pass and reorders the characters according the bidi algorithm and the characters' directionality attributes. This stream of reordered characters serves as input to the first positioning pass.

Every Unicode character has a "directionality" property that is used to determine its behavior with respect to the bidirectional algorithm. Within Graphite, the value of this property is accessible as a system-defined glyph attribute, `directionality` (abbreviated `dir`). If you wish to override the directionality of a character, or set it for a PUA character, you can do so by setting the `dir` glyph attribute in the glyph table. There is also a `dir` slot attribute (initialized from the glyph attribute) that can be set by means of rules.

The following are the most common `directionality` attribute values:

- `DIR_LEFT`: left-to-right (L)
- `DIR_RIGHT`: right-to-left (R)
- `DIR_ARABIC`: Arabic (AR; there are some subtle differences between `DIR_ARABIC` and `DIR_RIGHT`)
- `DIR_ARABNUMBER`: Arabic number (AN—left-to-right embedded in right-to-left)
- `DIR_WHITESPACE`: whitespace
- `DIR_OTHERNEUTRAL`: neutral

If you have a script that you know does not contain bidirectionality, you can indicate that the bidi pass should not be included by including the following global command:

```
Bidi = false;
```

This is an optimization and should have no effect on the rendering. In fact, the default value of the `Bidi` global setting is in fact false, so bidirectionality must be turned on for fonts where it is needed.

Exercise 15

Write a program to treat lowercase letters as left-to-right and uppercase as right-to-left (`DIR_RIGHT`). Do this by setting the `directionality` glyph attribute.

Hint: what is the minimum number of rules that you need?

Exploring user interface issues with bidirectional text

Note: The following tests require WorldPad.

Observe how insertion-point and range selections behave with mixed-direction text in WorldPad. What happens to white space at the line boundaries?

Change the paragraph direction to right-to-left; what differences do you observe?

Use the Tools-Options item to change from logical to visual arrow key behavior. Now how do the arrow keys operate?

Unit 16: Reordering

Now we return to the substitution table, to discuss yet another behavior it can be used to implement. This is reordering, specifically of the sort that we find in many scripts of South and Southeast Asia.

Here we extend the use of the @ symbol, which we saw briefly in our discussion of ligatures. The @ symbol is used to access an item in a rule. The number following the @ symbol indicates the position of the referenced slot with respect to its position in the context (the @ symbol alone indicates the slot itself).

```
gA  gB  >  @3  @1  /  _  gX  _;
```

The rule above replaces the A with the item at position 3 (the B) and replaces the B with the item at position 1 (the A) when an X occurs between the two characters. In other words, the A and the B switch places. Notice that the slot number for gB is not 2, but 3, its position within the entire context.

The @ symbol can also be used to copy a glyph with no changes, or with changes only to the slot attributes.

Exercise 16

Write a program to reorder back rounded vowels (o and u) such that they occur before a preceeding consonant. (Remember that reordering is handled in the substitution table.) You may ignore the problem of vowel sequences (unless you feel particularly ambitious).

Unit 17: Optional items

It is possible to specify optional items in a rule. This is done by following the optional item with a question mark. For instance, if we say that the intervening X in the rule from Unit 16 is optional, our rule looks like this:

```
gA  gB  >  @3  @1  /  _  gX?  _;
```

Notice that the indexing of elements must take into account all optional items; hence gB is considered item #3 in the above rule.

Optional markers can be placed either in the left-hand side or the context, but not in the right-hand side.

Ranges of items can be marked optional by using square brackets:

```
gA  gB  >  @3  @1  /  _  [gX gY gZ]?  _;
```

Note that in the rule above, the entire sequence 'X Y Z' must occur or none of it is allowed. You can also embed sequences of optional items. In the following rule, X may occur zero to three times:

```
gA  gB  >  @3  @1  /  _  [gX [gX gX? ]? ]?  _;
```

Exercise 17

Extend your program from Exercise 16 to handle the presence of a consonant cluster before the vowel to be moved; i.e., allow two optional consonants between the initial consonant and the vowel.

Hint: instead of switching items, you may need to delete the vowel and reinsert it.

Exploring Graide: the numbering of rules with optional items

Test your program in Graide with data the includes the optional consonants and data that does not, for instance: so sto stro to tro. Look at the list of rules in the Rules tab. What do you notice about the numbering of the rules?

Even though there is a single GDL rule that handle the various optional items, the Graphite compiler actually creates multiple rules corresponding to the presence or absence of the optional items. This is reflected in the numbering of the rules in the Rules tab.

Unit 18: Corresponding classes revisited

In Unit 4 we saw that Graphite uses the mechanism of corresponding classes to select glyphs for output. In other words, the members of the input and output classes are assumed to correspond, and the glyph to be output is the one that is at the same position within the output class as the input glyph is within the input class.

This idea can be extended to glyphs that are in different slots in the rule. The dollar sign ($) followed by a slot number indicates which slot in the input is to be used for the purposes of determining the relevant item position.

For instance, suppose in your font you have different versions of a certain lower diacritic depending on whether it is to be positioned below a base glyph with a descender or without. We can use the corresponding-class approach by creating two classes of glyphs, one containing the list of base glyphs and another containing the corresponding diacritics.

```
table(glyph)
      clsBase = unicode(0x61..0x7a);
      clsDiacPositioned = (gNormal, gNormal, gNormal,   // abc
                  gNormal, gNormal, gNormal, gLowered,  // defg
                  gNormal, gNormal, gLowered, …);       // hij…
endtable;
```

The two instances of `gLowered` in the `clsDiacPositioned` class correspond to the characters g and j, which have descenders.

Then the following rule is used to select the proper form of the diacritic, depending on the preceeding base character:

```
clsBase  clsDiac   >   @1  clsDiacPositioned$1;
```

Suppose the base glyph in the input is g. The `$1` syntax tells us to determine which item g is within its input class (`clsBase`). The answer is 7, and so the seventh item of class `clsDiacPositioned`—`gLowered`—is selected as the glyph to put into the output in the place of the original diacritic. On the other hand, suppose the base glyph in the input is d. Again, the `$1` syntax tells us to determine which item d is within the `clsBase` class. The answer is 4, and so the fourth item of class `clsDiacPositioned`—`gNormal`—placed in the output.

In both cases, `@1` causes the base character itself to be passed through unchanged.

Exercise 18

Extend your program from Exercise 17 to adjust the capitalization of the reordered items. If the original consonant was uppercase, the reordered vowel should be uppercase in the rendering, and the consonant should become lowercase. In other words, “Cup” is rendered as “Ucp”.