Smith Manual v0.3-dev, April 2015

Contents

1 Introduction

Smith is an extension to waf that is designed to create a build environment for the creation of Writing System Implementations. Such implementations are made up of a number of key elements: fonts, keyboards, sort orders.

1.1 Installation

Installing smith is simple. It consists of copying the smith file to a suitable directory, usually the project directory where it can be shared within any version control system. In addition a file `wscript` needs to be created to control the build process. This wscript file is in fact a python program but the way it is run is designed to hide that as much from the unsuspecting user as possible. Thus only those who want to get into the programming aspects need to.

In terms of necessary tools that smith makes use of, the details are listed in the parts of documentation specific to the tools that get used.

1.2 Execution

The heart of the build system is the `wscript` file that controls the build process. This is done by the python program creating a set of wsi component objects. The system then takes these objects and allows the user to run various build commands.

waf, on which smith is built, works by creating a build directory into which all the results are stored. This leaves the source directories pristine and makes for easy clearing up. The build directory is created using the command:

```
smith configure
```

This process creates the build directory, checks that all the tools that smith needs to achieve the build as described in `wscript` are available, and sets up various internal environment variables. Thus if any changes are made to the `wscript` that indicate what extra tools are needed, then the `configure` command needs to be rerun.

After configuration it is possible to build the system. This is done using:

```
smith build
```

This creates the final release forms of the various components that are to be built. For example, it will create any fonts or keyboards. But it does not create any installers, these need another command:

```
smith exe
```

This creates the installers described by the various package objects. In turn it also builds everything as specified by `smith build`.

```
smith pdfs
```

This creates font tests output as pdf.

```
smith fret
```

If the wscript has been set up to generate fret files, this will generate those files.

```
smith graide
```

This creates a subdirectory called graide that contains one .cfg file per font for use with graide. If the font has no graphite smarts, no configuration file is created (for obvious reasons).

```
smith svg
```

This creates svg test results for fonts.

```
smith zip
```

Creates a zip appropriate for a source package.

```
smith clean
```

Removes the various files created by `smith build` in the build directory.

```
smith distclean
```

Removes the build directory completely.

1.3 Writing wscript

The `wscript` file is a python program, but the internal program environment is set up to minimise the amount of actual programming that needs to be done. There is no setup needed in the file, and object registration is automatic. It is possible to add waf specific extensions to the file and for details of this, users should read the waf manual.

The basic process of describing a build process is to create writing system component objects. These objects are `font()`, `kbd()` and `package()`. Specific details on what information each of these objects requires is given in the corresponding sections of this document. Likewise examples are given in the sections.

The build process is about creating files from other files. Most of these processes are internal to the object, but it is possible to do some advanced configuration allowing the `wscript` writer to take more control over the build process. The functions described here should be considered advanced, and the beginning authors should not need to concern themselves with them initially.

cmd(cmd, [input files], **options)

The `cmd()` function specifies are command to run as a string, then a list of dependent input files that are referenced via `${SRC}` in the command string. The target (which is given by the context of `cmd()` function is accessible via the `${TGT}` string. The first parameter to the function is the command string to execute, which is executed from the build directory. There are various options that can be added to a `cmd()`:

late

If set to non zero, this says that the command should be executed as late in the sequence of commands to be run on a file as possible.

targets

This is a list of extra targets that this command generates. So a single command can create more than one file.

shell

If set, says that the command should not be broken on spaces into elements to pass to an exec call, but to be passed through the shell for shell processing. Use this if you use file redirection, for example.

create()

The `create()` function takes an initial parameter of the filename of the file to be created. The next parameter is a command to create the specified file. Usually this is a `cmd()` function. For exasmple, consider a processing path on an input font:

```
source = create('xyz.sfd', cmd('myfirstprocess ${SRC} ${TGT}', ['infile.sfd']),
                           cmd('mysecondprocess ${DEP} ${TGT}'))
```

process()

This function does an in place modification of the first parameter file that is assumed to already exist. The remaining functions are used to process the file in place. Often this is a `cmd()` function, but some other file type specific functions do exist. For details of them, see the relevant component type section. To reference the temporary input file referenced, use `${DEP}`

```
target = process('outfile.ttf', cmd('ttfautohint ${DEP} ${TGT}'))
```

When `process()` is used on a source file, smith has to think a little harder. smith works to a strict rule that no files are created or changed in the main source tree. This means that smith cannot change a source file in its original position. For similar reasons (which file should one read?), smith does not allow there to be an identically named file with the same path

in the source tree and in the build tree. So we can't simply copy the source file into the build tree and work on it there. Instead, smith creates a copy of the source file in the buildtree by stripping its path component and storing it in the `tmp/` directory. It then processes that in place. For the most part authors do not have to consider this, and using `process()` on a source file will *just work*. But there are rare situations where knowledge of the underlying actions are necessary.

Parameters for this function are:

nochange
If set, tells the system that there is no need to copy the dependency file before running the task. This is an internal parameter that users are very unlikely to need to use.

test()
This function applies a process to its output file with no expected output, so any `cmd()` would only have a `${TGT}` in the string. Of course other dependent inputs may be used. This is used for running files through checking processes that can fail, and give reports.

2 Tutorial

In this tutorial we will examine a number of wscript files. The first section is the largest and builds up a complex font creation scenario from humble beginnings.

2.1 Font Creation

2.1.1 font1 - Simplicity

We start with a simple, single font.

```
1 fontbase = '../../script-test/fonts/thai/'
2 font(target = 'Loma.ttf',
3      source = fontbase + 'font-source/Loma.sfd')
```

Due to the way the tutorial is structured, we keep the shared source files in a different directory tree. Line 1 sets up where that tree is.

In line 2, we create a new font object that we want the system to build. We specify the target filename. This file will be created in the build tree (`buildlinux2` on Linux but can be overriden to something like `results` by setting the out variable). Line 3 specifies where to find the source file. Notice that the target file is a `.ttf` file while the source is a `.sfd` file. Smith will use the necessary commands to convert from one to the other.

With this as our `wscript` file, we can build our font:

```
smith configure
```

This is the first step in building any project. This command tells smith to set up the build environment and search out all the programs that it may need for the various tasks we may ask of smith. If a necessary program is missing smith will stop at that point and indicate an error. Some programs are not strictly necessary and smith can run with reduced functionality without them. Such missing programs are listed in orange. All other programs that smith searches for and finds are listed, along with their locations, in green. So you can see exactly which program smith will use for any particular task. This is hepful especially in cases where you may have a locally self-compiled version: you can more easily see if smith has found the version in /usr/local/ instead of the stock packaged version.

```
smith build
```

This command tells smith to go and build all the objects the `wscript` says to be built. In this case just the simple `Loma.ttf` which will appear in `buildlinux2`. Not very exciting, but a good start.

2.1.2 font2 - Multiple fonts

Most font packages consist of more than one font file and this project is no exception. Can we scale our project to handle more than one file?

```
1 fontbase = '../../script-test/fonts/thai/'
2
3 for ext in ('', '-Bold') :
4     font(target = 'Loma' + ext + '.ttf',
5          source = fontbase + 'font-source/Loma' + ext + '.sfd')
```

This example shows the power of integrating a description with a full programming language. `wscript` files are python programs, albeit very enhanced ones. So we can use any python type constructs we might need. Usually the need is slight, and we show a typical example here.

Line 3 is the start of a loop. The lines below that are indented within the loop will be repeated for each value in the list. The first value is nothing (well the empty string ″) and the second is `-Bold`. Each time around the loop, the variable `ext` is set to the appropriate string. We will then use that variable to help set the appropriate values in the two font objects we are creating.

Each time around the loop, we create a new font object. In line 4 we create a new font object whose target font file's name is dependent on the `ext` variable which is set to the various strings from the list at the start of the loop. So we will end up creating two fonts. One called `Loma.ttf` as before, and one called `Loma-Bold.ttf`. Line 5 gives the source files for each of these fonts.

It may seem easier just to expand out the loop and have two `font()` object commands, but as the complexity of this `font()` grows, we will see the value of using a loop. The advantage of adding the loop early is that we can make appropriate use of `ext`.

Now when we come to build this project, we will get two fonts:

```
smith configure
smith build
```

2.1.3 font3 - Packaged

It's good that we can create multiple fonts, but what do we do with them then? There are two typical products that people want from a font project: a `.zip` file containing the fonts and a `.exe` Windows installer that allows someone to simply run the program to install the fonts. Smith can create these two products from a wscript, but it needs just a little more information to do so:

```
1 APPNAME = 'loma'
2 VERSION = '0.0.1'
3 fontbase = '../../script-test/fonts/thai/'
4
5 for ext in ('', '-Bold') :
6     font(target = 'Loma' + ext + '.ttf',
7          source = fontbase + 'font-source/Loma' + ext + '.sfd')
```

Line 1 gives the base name of the products that will be created and line 2 gives the version of that product. Notice that the version variable is a string and does not have to be numeric. Case is important here, these are, in effect, magic variables we are setting that smith looks up.

To build this project, we do the same as before, but we can also use two extra commands:

```
smith configure
smith build
smith zip
smith exe
```

`smith zip` will create `loma-0.0.1.zip` in the `buildlinux2` subdirectory. This zip file contains the two target fonts the build created. `smith exe` creates a Windows installer, and in keeping with naming conventions for installers, the APPNAME has been title cased to produce `Loma-0.0.1.exe`. We will go into much more detail on packaging in the packaging tutorial section.

2.1.4 font4 - Internal processing

Before our example gains smart font support and grows in complexity, there is one area of control that is worth examining. For the most part, when creating a `wscript` one fills in the various *forms* that create the objects, and smith knows what needs to happen to make things turn out right. But while this makes for pretty tutorials, real world projects have unique quirks that require the ability to add commands into the processing or to create things dynamically. In this exercise we will add a process to the source font:

```
1  APPNAME = 'loma'
2  VERSION = '0.0.1'
3  prjbase = '../../script-test/'
4  fontbase = prjbase + 'fonts/thai/font-source/'
5  rmoverlap = prjbase + 'bin/rmOverlap'
6
7  for ext in ('', '-Bold') :
8      fbase = 'Loma' + ext
9      font(target = fbase + '.ttf',
10          source = process(fontbase + fbase + '.sfd', cmd('../' + rmoverlap + ' ${DEP} ${TGT ↩
               }')))
```

The interest lies in line 10. Here we use a `process()` function to tell smith that we want it to run a command over the source font before converting it to a `.ttf`. A `process()` function takes a file which already exists (either in the source tree or one that is generated by another process) and then runs the list of `cmd()` function results over it in order. In this case the command is to run a script that removes overlap from all the glyphs in the font. The command string takes some study. The program takes two command line parameters, an input font file and an output font file. We represent these in the command string by `${DEP}` (the dependent file) as the input and `${TGT}` as the output file. smith will fill these in appropriately when it comes to run the command. In addition, note the initial `'../'` at the start of the command string. This is because all commands in smith are run from the `buildlinux2` directory and so we have to go up one level to get back to the project root where the `wscript` file is and then from there we can navigate to the actual remove overlap script.

The rest of the new lines in this exercise are simply extra variables being used to make the file easier to read, otherwise some of the lines would become excessively long and confusing. Notice that all the magic variables in a `wscript` that smith considers are all caps. That is if you use a variable name with a lowercase letter in it, you are sure to be safe from smith assuming some special meaning to that variable.

For the most part we are not very interested in precisely what smith is doing to get the results we want. But sometimes it helps to know, and all that cryptic output streaming by isn't much help. But there is a way to get something more helpful. First we need to get back to a completely pristine source tree:

```
smith distclean
```

Now we can configure and run in a way that has smith tell us what it is doing:

```
smith configure
smith build -j1 -v
```

In my case, here were my results:

```
$ smith build -j1 -v
smith: Entering directory `/tutorial/tutorial/font4/buildlinux2'
[1/6] tmp/Loma.sfd: ../../script-test/fonts/thai/font-source/Loma.sfd -> buildlinux2/tmp/ ↩
    Loma.sfd
16:56:00 runner ['cp', '../../../script-test/fonts/thai/font-source/Loma.sfd', 'tmp/Loma. ↩
    sfd']
[2/6] tmp/Loma-Bold.sfd: ../../script-test/fonts/thai/font-source/Loma-Bold.sfd ->  ↩
    buildlinux2/tmp/Loma-Bold.sfd
16:56:00 runner ['cp', '../../../script-test/fonts/thai/font-source/Loma-Bold.sfd', 'tmp/ ↩
    Loma-Bold.sfd']
[3/6] tmp/Loma.sfd[0]../../../script-test/bin/rmOverlap:
16:56:01 runner /tutorial/tutorial/font4/buildlinux2/tmp/Loma.sfd-->/tutorial/tutorial/ ↩
    font4/buildlinux2/.tmp/tmp/Loma.sfd
```

```
16:56:01 runner ' ../../../script-test/bin/rmOverlap .tmp/tmp/Loma.sfd tmp/Loma.sfd '
[4/6] tmp/Loma-Bold.sfd[1]../../../script-test/bin/rmOverlap:
16:56:01 runner /tutorial/tutorial/font4/buildlinux2/tmp/Loma-Bold.sfd-->/tutorial/tutorial ↩
    /font4/buildlinux2/.tmp/tmp/Loma-Bold.sfd
16:56:01 runner ' ../../../script-test/bin/rmOverlap .tmp/tmp/Loma-Bold.sfd tmp/Loma-Bold. ↩
    sfd '
[5/6] Loma.ttf_sfd: buildlinux2/tmp/Loma.sfd -> buildlinux2/Loma.ttf
16:56:01 runner " /usr/bin/fontforge -lang=ff -c 'Open($1); Generate($2)' tmp/Loma.sfd Loma ↩
    .ttf "
Copyright (c) 2000-2012 by George Williams. See AUTHORS for contributors.
 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
 with many parts BSD <http://fontforge.org/license.html>. Please read LICENSE.
 Executable based on sources from 02:55 UTC  4-Dec-2013-ML-D.
 Library based on sources from 02:55 UTC  4-Dec-2013.
 Based on source from git with hash:
[6/6] Loma-Bold.ttf_sfd: buildlinux2/tmp/Loma-Bold.sfd -> buildlinux2/Loma-Bold.ttf
16:56:01 runner " /usr/bin/fontforge -lang=ff -c 'Open($1); Generate($2)' tmp/Loma-Bold.sfd ↩
     Loma-Bold.ttf "
Copyright (c) 2000-2012 by George Williams. See AUTHORS for contributors.
 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
 with many parts BSD <http://fontforge.org/license.html>. Please read LICENSE.
 Executable based on sources from 02:55 UTC  4-Dec-2013-ML-D.
 Library based on sources from 02:55 UTC  4-Dec-2013.
 Based on source from git with hash:
smith: Leaving directory `/tutorial/tutorial/font4/buildlinux2'
'build' finished successfully (0.919s)
```

Thankfully, on a modern terminal, the colourising helps makes more sense of the voluminous information. But it is helpful once you learn to read it. The timestamped `runner` lines give the precise command lines that are run at each stage of the build.

Clearly the key to getting this output is in the command-line options to `smith build`. The `-v` says to output the extra information. But since smith tries to use multiple processors if you have them, to speed up the build (for example without the `-j1`, my build runs in 0.479s), it means the output can get interleaved. It is therefore wise to restrict smith to a single process while outputting this information, and this is done using `-j1`.

2.1.5 font5 - Smarts and Basic Tests

In this exercise we grow our description to add OpenType and Graphite tables and also add some tests.

```
prjbase = '../../script-test/'
fontbase = prjbase + 'fonts/thai/font-source/'
rmoverlap = prjbase + 'bin/rmOverlap'

APPNAME = 'loma'
VERSION = '0.0.1'
TESTDIR = prjbase + 'fonts/thai/tests'
TESTRESULTSDIR = 'tests'
TESTSTRING = u"\u0e01=chicken, \u0e02=egg. Problem solved"

for ext in ('', '-Bold') :
    fbase = 'Loma' + ext
    font(target = fbase + '.ttf',
         source = process(fontbase + fbase + '.sfd', cmd('../' + rmoverlap + ' ${DEP} ${TGT ↩
             }')),
         opentype = internal(),
         graphite = gdl(fbase + '.gdl', master = fontbase + 'master.gdl'),
         ap = fbase + '.xml',
         script = 'thai',
         fret = fret(params = "-r")
        )
```

Line 15 tells smith how the OpenType tables will be generated for this font. It is possible to compile in VOLT tables, or, as here to use the internal description already in the font. The `internal()` does very little, but it does indicate to smith that the font has OpenType tables and that they should be tested.

Line 16 tells smith how the Graphite tables are to be added. There is currently only one form for Graphite source, and that is GDL. The `gdl()` object tells smith how to generate and bring together the various files that typically make up a Graphite description. A typical Graphite project has an autogenerated component (which is the first parameter to the gdl() (`fbase+'.gdl'`) and a common core gdl file that is hand authored (the `master` parameter). Smith then does the work to generate the files and compile them into the font.

Line 17 talks about an attachment point database. This file holds information about glyphs in the font that cannot be held by TTF. Most importantly this file holds the positions of anchor points on the glyphs, and these positions are used when autogenerating smart code, either for volt() or gdl() or whatever. Depending on the source file format, the file may be autogenerated or be required as part of the source files.

These three lines are all it takes to add a sophisticated smart font build system to the font creation. The rest of this section will look at basic font testing.

The basic principle of font testing in smith is that there is a test directory containing test data. This data is then applied to the various fonts and results are generated in the build tree. Test data can be of various formats, but the easiest to work with is simple `.txt` files that are treated as one paragraph per line files.

Line 7 gives the directory where the test files may be found. Since it is outside the project tree rooted in the directory containing the wscript, we have to specify where in the buildtree we want the test results to be put. Line 8 specifies that subdirectory.

Smith allows for user defined tests, but there are some defaults built in, which we will examine here.

```
smith pdfs
```

This tells smith to generate pdfs of each test file for each font for each smart font technology. That's quite a few for each, but it means that you can look at any particular font and its smart rendering technology for each test. The files end up in `buildlinux2/tests` based on the value of `TESTRESULTSDIR`.

So for example, the test file riwordsu-short.txt will generate 4 pdf files: for the regular font: riwordsu-short_loma_ot.pdf, riwordsu-short_loma_gr.pdf and for the bold font: riwordsu-short_loma-bold_ot.pdf and riwordsu-short_loma-bold_gr.pdf. The `_ot` extension is used for OpenType rendered texts and `_gr` for the graphite rendered texts. The texts are rendered using XeTeX.

The other file in the tests directory is `patani-malay.htxt`. The `h` in htxt tells smith to preprocess the file to convert strings of the form \uxxxx into the corresponding Unicode character before rendering. This makes it easier to create test files.

Line 18 is an important line for OpenType testing since it specifies which script to use when running the OpenType shaping engine.

Another aspect of testing is regression testing. Can we find out what has changed between this font and a known good version? The way this works is that we store known good versions of the fonts and then have smith run tests against both fonts and compare the results. The default directory to keep the font files in is `standards/`.

```
smith test
```

The results end up in the `RESULTSDIR/regression` directory as `.log` files. If there are no differences, the files are 0 length.

A further target that is useful is the ability to create font reports that show all the glyphs in a font. We set this as a font product rather than a kind of test, in line 19. The default target filename is the same as the `target` .ttf file but with a .pdf extension instead. The file is built as part of `smith build`.

There are two other targets that this wscript enables:

```
smith waterfall
smith xfont
```

Line 8 specifies a string that will be used in creating the waterfall files and also the cross font summary files. `smith waterfall` creates one file per font and technology and stores it in the `waterfalls` sub directory of the `TESTRESULTSDIR`, prefixing each font and technology with `Waterfall`. `smith xfont` creates one file per technology in the `TESTRESULTS`

`DIR` called `Crossfont_ot.pdf` or `Crossfont_gr.pdf` that contains the test string output with the font name, one per line.

Another feature of smith is its ability to integrate with `graide`. Graide is a graphically based IDE for developing GDL Graphite source code. It also incorporates a Graphite debugger to help font developers see how their code executes.

```
smith graide
graide -p graide/Loma.cfg
```

Running `smith graide` causes smith to create graide configuration files in a `graide/` subdirectory. This is one of the few commands that creates files outside of the buildlinux2/ tree. The user can then run graide referencing one of these configuration files. One file is made per font.

A word of advice. Since, most often, smith does not generate .gdx files when it runs grcompiler (.gdx files are grcompiler debug files), it is best to recompile the font on loading into graide.

The configuration is designed to restrict graide to just editing GDL. If you want to use graide to adjust attachment points or add them, then you will need to enable writing to the AP.xml, in the graide configuration, and you are then responsible for propagating those changes back from the AP.xml to your source font.

2.1.6 font6 - Metadata

So far we have concerned ourselves with the mechanics of font creation. But in order to release a font package we also need to concern ourselves with the metadata that is involved in producing a font release.

```
1  prjbase = '../../script-test/'
2  fontbase = prjbase + 'fonts/thai/font-source/'
3  rmoverlap = prjbase + 'bin/rmOverlap'
4
5  APPNAME = 'loma-minority'
6  VERSION = '0.0.1'
7  TESTDIR = prjbase + 'fonts/thai/tests'
8  TESTRESULTSDIR = 'tests'
9  TESTSTRING = u"\u0e01=chicken, \u0e02=egg. Problem solved"
10 DESC_SHORT = "Loma with minority extensions"
11 DESC_LONG = """Loma with minority extensions
12 These fonts support extra characters and sequences as needed for
13 minority languages that use the Thai script. Specifically this
14 font supports the following language needs: Patani Malay, So"""
15 LICENSE = "GPL.txt"
16
17 for ext in ('', '-Bold') :
18     fbase = 'Loma' + ext
19     font(target = process(fbase + '-Minority.ttf', name("Loma Minority")),
20          source = process(fontbase + fbase + '.sfd', cmd('../' + rmoverlap + ' ${DEP} ${TGT ↩
               }')),
21          version = 0.1,
22          woff = woff(),
23          opentype = internal(),
24          graphite = gdl(fbase + '.gdl', master = fontbase + 'master.gdl'),
25          ap = fbase + '.xml',
26          script = 'thai',
27          fret = fret(params = "-r")
28         )
```

While we have used an existing font: Loma as our base font, we can't really release a new version of Loma since that is a font owned by someone else. The font has a GPL license and so we are free to develop our own version of Loma under that license. It makes sense, therefore, to change the name to something more appropriate: `Loma Minority`. We do this in a number of places in the wscript: Line 5 changes the installer application name. Line 19 changes both the name of the font file generated (and all derived products), but it also processes that font file to change the internal name to "Loma Minority", using a process() and a name() function that acts like a cmd() that is suited to font renaming.

We also enhance the installation package to include a short and long description. We also include a license file, on Line 15, for the installer that reflects what is in the font. Smith also provides a mechanism to directly support the OFL license inside the font and font package, but this is not relevant here.

We also set the version of the font inside the font using a `version` parameter on line 21. This means that smith can set the version inside the font if so desired. This saves editing source fonts all the time to account for what is a build parameter.

The Web Open Font Format (WOFF) is designed particularly for distribution of web fonts and smith can generate such files from the target .ttf font file. The default parameters for this object take the font target filename as the basis of the woff filename, which is sufficient for our needs.

2.1.7 font7 - More Tests

This section is for those interested in doing more advanced types of testing. For most projects there is no need to go to this level of complexity and many users never need to use these capabilities. So this exercise has been placed after the exercise that pretty much completes font creation. We also try to introduce as many advanced techniques as we can, even if the results end up being a little contrived.

Font testing is not limited to just the inbuilt test types. Smith supports the integration of other test programs as you the user desires, so long as they are command line based, non interactive and report generators.

```
1  prjbase = '../../script-test/'
2  fontbase = prjbase + 'fonts/thai/font-source/'
3  basebin = prjbase + 'bin'
4
5  APPNAME = 'loma-minority'
6  VERSION = '0.0.1'
7  TESTDIR = prjbase + 'fonts/thai/tests'
8  TESTRESULTSDIR = 'tests'
9  TESTSTRING = u"\u0e01=chicken, \u0e02=egg. Problem solved"
10 DESC_SHORT = "Loma with minority extensions"
11 DESC_LONG = """Loma with minority extensions
12 These fonts support extra characters and sequences as needed for
13 minority languages that use the Thai script. Specifically this
14 font supports the following language needs: Patani Malay, So"""
15 LICENSE = "GPL.txt"
16
17 t = fonttest(targets = {
18     'pdfs' : tex(),
19     'test' : tests(),
20     'report' : tests({'report' : cmd("${FONTREPORT} ${SRC[0]} > ${TGT}")}, coverage = " ↩
           fonts")
21     })
22
23 for ext in ('', '-Bold') :
24     fbase = 'Loma' + ext
25     font(target = process(fbase + '-Minority.ttf', name("Loma Minority")),
26          source = process(fontbase + fbase + '.sfd', cmd('${RMOVERLAP} ${DEP} ${TGT}')),
27          version = 0.1,
28          woff = woff(),
29          opentype = internal(),
30          graphite = gdl(fbase + '.gdl', master = fontbase + 'master.gdl'),
31          ap = fbase + '.xml',
32          script = 'thai',
33          fret = fret(params = '-r'),
34          tests = t
35         )
36
37 def configure(conf) :
38     import os
39     basepath = os.environ['PATH'].split(os.pathsep) + [os.path.abspath(basebin)]
40     conf.find_program('rmOverlap', path_list = basepath)
```

```
41      try :
42          conf.find_program('fontreport', path_list = basepath)
43      except conf.errors.ConfigurationError :
44          pass
```

The interesting section is in lines 17-21. These lines create a fonttest object that is then referenced within the font at line 34. A fonttest object adds new smith commands. This example adds the three smith commands: pdfs, test and report. Notice that the `smith pdfs` command is actually implemented using a fonttest() object. The `targets` parameter to fonttest uses a python data structure called a dictionary. This is indicated by the `{` at the start (and `}` at the end). Dictionary elements consist of a string before a `:` and a value after it. The value before the `:` is known as the key and the value after as the value. So a dictionary is set of key, value pairs. In our case, the keys here indicate smith commands and the values are the test objects that get executed for the command.

The first two commands use default test objects appropriate to the type of command. The `pdfs` command executes a `tex()` object that does all the xetex processing of test files. Likewise the `test` command executes a default `tests()` object which implements the regression testing.

Our new command `report` also uses a `tests()` object. But in this, we give another dictionary of key, value pairs. The key is a subdirectory under the TESTRESULTSDIR and the value is a `cmd()` object that gives the command to execute. In this case we are running the fontreport program. The reference to '${SRC[0]}` says to use the first element from the inputs. The inputs has 3 elements: the font, the text file to test and the corresponding `standards/` font file. We only need the first of these and list indices all start from 0 in python. In addition, we use the parameter `coverage` to say that we only want to run tests one per font, and not one per test file per shaper per font. The `> ${TGT}` says that the output that the program produces, which would normally be printed on the screen is to be sent to the target log file instead.

Another thing we have changed is that rather than hardwiring various of the specialist programs into our wscript, we now will get smith to go and search for them. At line 37, we introduce another new python concept: the function. Each smith command will search for a function in your wscript with the same name as the command and will execute it. For more information of what to do then you should read the manual for the underlying framework that smith is built on, which is waf. The variable passed to us is a waf context that can be used to do various things like add commands to the build process, etc. In our case we want to have smith search for various programs. `find_program()` is the key that tells smith to search for the programs. In the case of `rmOverlap`, that is necessary for the build, so if it is missing we want the configuration to fail. But in the case of the test script, we only lose the ability of that one test type if the script is missing, so we don't want to fail the configuration. This is a marginal call, but we do at least get to see the pattern for achieving this (lines 41-44). In each case find_program() takes a list of paths to search, and it only searches those directories, not directories below those, unless explicitly listed.

3 Directory Structure

Having a good layout for all the different files in a script project is very helpful.

A reference template structure is in preparation, for now the following structure description is provided:

3.1 Create Working Area

If you need a suggestion of where to place all the needed files, the following directory structure is suggested.

Create a directory to work in. We will use the directory `projects\`. You can call this directory whatever you want and put it wherever you like, just remember to adjust the following instructions accordingly. Your home folder is a good place for this directory.

In `projects` create a directory called `script`. In `fontwork` place Perl sources (either from CPAN tarballs or subversion repositories) and other miscellaneous files. In `script` there should be one directory for each script. The directory name should be the ISO 15924 four-letter code for the script you are working with. For example `deva` for Devanagari, `latn` for Latin (or Roman) script.

3.2 Script Projects

These projects are named for the ISO 15924 four letter codes mentioned above. While it isn't necessary to follow the script project directory structure when using the font template build system, it is a useful directory structure.

fonts/
Contains one directory per font project in this script

keyboards/
Contains one directory per keyboard layout collection

mappings/
Contains one directory per legacy encoding. Inside that is any mapping tables and a sample font in that encoding.

wsis/
Each directory under here corresponds to a particular language that uses script. The directory name is the RFC4646bis code for the language. Each language directory in its turn contains other directories:

words/
This directory contains Unicode encoded wordlist files

cldr/
This directory contains any LDML files for this language in this writing system

toolbox/
This directory contains any Toolbox `.lng` files for this language and script in Unicode. Non-Unicode files are stored under a corresponding directory in the appropriate `mappings` directory.

texts/
Contains sample texts in the language, including `.pdf` to show how the text should be rendered.

4 Fonts

Fonts form the heart of the build system, given that they are the most complex component type to create and work with.

The minimum attributes a font object needs are: `target` and `source`. For example, the following `wscript` file is about as simple as one can get:

```
font(target = 'results/myfont.ttf',
     source = 'myfont.ttf')
```

This short file does more than might be expected. First of all it copies the input file `myfont.ttf` to an output file `build/results/myfont.ttf`footnote:[We will use unix style `/` for path separators]. This copy may seem redundant, but it is necessary for the rest of the system to work, and not all source fonts are unmodified `.ttf` files. It will also add this font to the default package, allowing a Windows installer to be created. If there are tests `.txt` files in a directory called `tests` then these can be run against this font.

Notice that an input and an output file may not have the same name. Even if the output file ends up in `build/` it still corresponds to a real input file that may or may not be in `build/`. So file paths must be unique if the files are unique.

What if the source isn't a `.ttf` file. We can simply change the above example to:

```
font(target = 'results/myfont.ttf',
     source = 'myfont.sfd')
```

and the system will automatically convert the FontForge source font to TrueType as it is copied into the `build` results directory tree. Here we wouldn't actually need the `results/` prefix to the target because the filename isn't the same as the `source` attribute.

The complete list of core attributes to a font are:

target
Output file for the generated font within `build`.

source
Basic design file used to generate the initial form of the output font.

version
This takes a version number in the form *x.y* (a floating point number) and sets the output font to have that version. It may also be a tuple of the form (x.y, "text") where the text will be appended to the version string inside the font.

sfd_master
This attribute specifies a FontForge file that should be merged with the source FontForge file when creating the target. If the sfd_master file is the same as the source, then sfdmeld is not run.

ap
Attachment point database associated with the source font.

classes
Classes .xml file that adds class information to the attachment point database before conversion into smart font source code.

copyright
Copyright string to insert into the font.

tests
Test object to use for test generation. If not specified, the global font test object is used.

package
Package object to insert this font into. If not specified the global package is used.

typetuner
Specifies that typetuner should be run on the target and to use the given file as the typetuner configuration xml file.

4.1 OpenType

There are two ways of adding OpenType information to a font. One is to already have it in the source font. This is the most common approach. We need to indicate to the font builder that we are working with an OpenType font, even if everything is internal to the font. The font builder needs to know for font testing purposes or if the font is generated from a legacy font.

```
font(target = 'results/myfont.ttf',
     source = 'myfont.sfd',
     opentype = internal())
```

This will generate tests pertinent to OpenType testing. See the section on font tests.

One approach sometimes used for FontForge based projects is to keep all the lookups in one font and then to share these lookups across all the fonts in a project. For this we simply specify a `sfd_master` attribute and the font builder will use `sfdmeld` to integrate the lookups in the master into each font as it is built. There is no need to specify that the font is OpenType in this case, because that is obvious from the context:

```
font(target = 'results/myfont.ttf',
     source = 'myfont.sfd',
     sfd_master = 'mymaster.sfd')
```

Obviously, if the `sfd_master` attribute is the same as the `source` file then no merge occurs. This is an alternative way of specifying that the font is OpenType.

Another approach to adding OpenType tables to a font is to use an external tool to create the lookups in and then to have that compile them into the font. Currently, the only external tool supported by the system is Microsoft's VOLT (Visual OpenType Layout Tool). This uses a command line VOLT compiler to integrate the `.vtp` source into the font. In addition, the `.vtp` source is autogenerated from a source and any other elements that go to make the final integrated source. For example we show a maximal `volt()` integration to show all the components and then discuss them.

Notice that while the initial parameter to such objects as `volt` is required, all named parameters are optional.

```
font(target = 'results/myfont.ttf',
     source = 'myfont.ttf',
     opentype = volt('myfont.vtp',
                     master = 'mymaster.vtp'),
     ap = 'myfont.xml',
     classes = 'project_classes.xml')
```

We define the `.vtp` file to create for this font which will then be compiled into the font using `volt2ttf` as `myfont.vtp`. We also declare a shared master volt project that is shared between all the fonts (well at least this one!). In building a largely automated volt project, a program `make_volt` is used that can take attachment point information from an xml database `myfont.xml`. This may be augmented with class information using `project_classes.xml`. These two file references are within the font rather than the volt details because they are shared with other smart font technologies particularly graphite.

The complete list of attributes to Volt() are:

master
: The volt source that is processed against the font to generate the font specific volt to be compiled into the font.

make_params
: These parameters are passed to the make_volt process. The value is a string of parameters.

params
: These parameters are passed to volt2ttf to modify the compiling of the volt source into OpenType tables.

no_make
: If this attribute is present, make_volt isn't run and the first parameter is assumed to be the specific .vtp for this font.

no_typetuner
: The VOLT2TTF program used to compile the volt into opentype, also has the capability to emit an XML control file for typetuner. By default, if the font requests typetuner be run, the volt2ttf options will be set to generate this file. Setting this attribute stops this generation from happening and you will need to create the file some other way.

More recently a textual representation of OpenType has been developed by Adobe. The Adobe Font Development Kit for OpenType (AFDKO) has a textual syntax for OpenType lookups, called a feature file. smith can work with these .fea files. It works by creating a font specific .fea file containing key classes and also attachment lookups for all the attachment points in the font. These can then be referenced in the appropriate features. It is unlikely that anyone will not use a master fea file.

```
font(target = 'results/myfont.ttf',
     source = 'myfont.ttf',
     opentype = fea('myfont.fea',
                    master = 'mymaster.fea'),
     ap = 'myfont.xml',
     classes = 'project_classes.xml')
```

The complete list of attributes to fea() follow those of other classes:

master
: The fea source that will be included at the end of the autogenerated .fea file.

make_params
: Extra parameters to pass to make_fea, the tool that is used to generate the dynamic .fea file.

keep_feats
: This tells the feature processor to keep all the lookups associated with a given feature that are already in the font, and not wipe them when merging the feature file. For example, keeping the kern feature lookups, which are often best handled in a font design application rather than in fea files.

no_make
: If this attribute is present, then make_fea isn't run and the first parameter references a file that already exists rather than one that will be created by fea().

4.2 Graphite

Adding Graphite tables to a font is much like adding VOLT information. The relevant files are declared either to the font or a `gdl()` object. For example:

```
font(target = 'results/myfont.ttf',
     source = 'myfont.ttf',
     graphite = gdl('myfont.gdl',
                    master = 'mymaster.gdl'),
     ap = 'myfont.xml',
     classes = 'project_classes.xml')
```

Notice that the `ap` and `classes` attributes have the same values and meaning as for OpenType tables. This is because the information is used in the creation of both sets of tables. The `myfont.gdl` is created by the `make_gdl` process and it pulls in `mymaster.gdl` during compilation.

The complete list of attributes to a gdl() object are:

master
Non-font specific GDL that is #included into the font specific GDL.

make_params
Parameters passed to `make_gdl`.

params
Parameters to pass to `grcompiler` to control the compilation of Graphite source to Graphite tables in the font.

no_make
If this attribute is present, make_gdl is not run and the first parameter is assumed to be the gdl for the specific font.

4.3 Legacy Fonts

Many fonts are actually built from another font, either legacy encoded or generated from a source font or fonts. This can be achieved by giving a `legacy()` object as the `source` attribute for the font. For example, for a font generated from a legacy font using `ttfbuilder` we might do:

```
font(target = 'results/myfont.ttf',
     source = legacy('myfont_src.ttf',
                     source = 'my_legacyfont.ttf',
                     xml = 'legacy_encoding.xml',
                     params = '-f ../roman_font.ttf',
                     ap = 'my_legacyfont.xml'))
```

The legacy object creates the source font that is then copied to the output and perhaps smarts are added too.

The complete set of attributes to a `legacy()` object is:

source
The legacy source font (`.ttf`) to use to convert to the Unicode source font.

xml
ttfbuilder configuration xml file to use for the conversion

params
Command line arguments to ttfbuilder. Note that files specified here need `../` prepended to them.

ap
Attachment point database of the legacy font that will be converted to the font.ap attribute file.

noap
Instructs the legacy converter not to create the ap database specified in the font. This would get used when another process, after legacy conversion, modifies the font and then you want the build system to autogenerate the ap database from that modified font rather than from the legacy font conversion process.

4.4 Licensing and Copyright

Fonts contain both copyright and license information. The `copyright` attribute is the copyright string for the font and if present that string will be inserted into the font as the copyright statement.

The license object is an object that describes how to generate the specific license for this font. It also allows the collecting of license information for all the fonts in a package. Currently, the only license type supported is the OFL. For example:

```
font(target = 'results/myfont.ttf',
     source = 'myfont.ttf',
     copyright = 'Copyright 2012, Acme <email|URL>',
     license = ofl('myfont', 'Acme',
                   version = 1.1,
                   copyright = 'Copyright 2012, Acme Labs <email|URL>',
                   file = "OFL.txt'))
```

The first parameter for the `ofl()` object is the list of Reserved Font Names (RFN). There may be more than one RFN. (For more details about the RFN mechanism please refer to the OFL FAQ: http://scripts.sil.org/OFL). These all appear in the license statement as the reserved font names for that particular version of the font project. In this example the name of the font and the name of the company of the author are reserved so that any derivative must be distinct from these two names. This is designed to allow all changes to the font but to prevent collision of fonts with different features but the same names. The goal is to reduce confusion for both font authors and font users. All the reserved font names in all the fonts in a package - various other authors can indicate their own reserved font names - are collected into one list that is then put into the license for the package. The OFL version may be given and defaults to 1.1. The copyright message is put at the start of the license statement. If not specified the global variable COPYRIGHT is used for this. The file where the license ends up may be given but defaults to `OFL.txt`.

4.5 WOFF

The WOFF format is a different file format based on TTF. Smith can generate .woff files. For example:

The `woff` object takes these attributes (which could well grow):

params
: This string is passed as the command line options to the ttf2woff command.

```
font(target = 'results/myfont.ttf',
     source = 'myfont.ttf',
     woff = woff('results/myfont.woff'))
```

4.6 Fret

Fret is a font reporting tool that generates a PDF report from a font file, giving information about all the glyphs in the font.

The `fret` object takes these attributes:

params
: A parameter list to pass to fret. If not specified, then fret is run with the `-r` command line argument.

```
font(target = 'results/myfont.ttf',
     source = 'myfont.ttf',
     fret = fret('results/myfont.pdf', params='-r -o i'))
```

4.7 Tests

smith includes a powerful system for testing with the potential to add different types of font tests. Currently there is only one font test type and also by default a single font test object is created that is used by all fonts for which no tests attribute is given.

There are two attributes associated with a font declaration that are to do with testing:

test_suffix
The way a font is identified as part of a test result filename is based on the target file for that font. If a font has a `test_suffix` attribute then this is used instead for the font identifying name.

tests
This is a fonttest object that describes the testing targets available for this font. If none is specified then a global shared default fonttest object. This default is described later.

The `fonttest` object takes these attributes:

testdir
This gives the directory relative to which all test files are to be found in the source tree. If unspecified, the global variable TESTDIR is used to give a default value. If this isn't set then the `tests` dir is used. So if one had tests in a different directory the following would work:

```
TESTDIR = 'test-suite'
font(target = 'results/myfont.ttf',
     source = 'myfont.ttf')
```

resultsdir
This specifies the directory under the build directory into which test results are placed. It can be set globally via TESTRESULTSDIR and defaults to the same directory as testdir.

texts
This is a glob string or list of glob strings that describe the text files to use for testing. The default is `*.txt`. For each text file and each font a `.tex` file is created that references the text file, the font, the script and language. In addition tests are run for each smart font technology (`gr` and or `ot`). These files are then generated to pdf.

htexts
Creating text files for complex scripts when there is no keyboard and at the start of a development project, can be problematic. One approach is to simplify the entry of unicode codepoints through the use of a string in the text file of the form `\u`*xxxx* which is a unicode scalar identifier (including support for supplementary plane data). Such files (defaulting to `*.htxt` are preprocessed into the corresponding `*.txt` and added to the list of `texts` test files.

targets
While the test system is designed to have sensible defaults, it is possible to override what happens when a particular test target is specified. This dictionary contains a test object associated with each of the test targets: pdfs, svg, test. The default test type for each target is pdfs : tex(), svg : svg(), test : tests().

extras
Often a project will want to add tests to the existing defaults, rather than overriding them. This parameter takes the same structure as a targets parameter but instructs the system to add the tests rather than replace the defaults with these.

In effect, the default behaviour for testing can be written entirely in wscript explicitly:

```
TESTDIR = 'test'
TESTRESULTSDIR = TESTDIR
globaltest = fonttest(targets = { 'pdfs' : tex(), 'svg' : svg(), 'test' : tests()},
                      texts = '*.txt',  # all these are defaults and can be omitted
                      htexts = '*.htxt'
                      testdir = TESTDIR,
                      resultsdir = TESTRESULTSDIR)
```

```
font(target = 'results/myfont.ttf',
     source = 'myfont.ttf',
     test_suffix = 'myfont',     # derived default
     tests = globaltest)
```

The various test target classes are described here with the various parameters to each.

4.7.1 tex

This class is used to generate PDF files from input test files. It takes the following parameters:

files
: This lists the files, relative to the testsdir, that should be used as sources for this test. The list is held in a dictionary with the value being key value pairs either stored as a dict or as a string with the pairs separated by &, as in key=value&key2=value2. These key values are used to set feature values and language and script for the file being tested. Any keys and values my be included. Particular keys are:

  lang
  : Specifies the language code to pass for rendering.

  script
  : Specifies the script code to pass for rendering

texs
: The difference with these `.tex` files (defaulting to `*.tex`) is that they are treated as simply TeX files to be run through XeTeX for test purposes. This gives the user greater control over complex text runs than a simple text file can give.

htexs
: These files are parsed for Unicode expansions before being treated as files in the `texs` list. Defaults to **.htex*.

4.7.2 crossfont

This class creates a single PDF file from all the fonts, with a given string output in the font next to the font file name. It takes the following parameters:

text
: The text string to be rendered in the font. If not included, it is taken from the global TESTSTRING, and failing that, is empty.

file
: Filename base (i.e. no extension), including path relative to the testsdir, where the output should be generated. Defaults to CrossFont.

size
: This is a number that specifies the font size in points. Defaults to the global TESTFONTSIZE (in points) and failing that to 12pt.

featstr
: Specifies the feature string that should passed to XeTeX after the script information. Each feature is separated from the next by a colon. The language feature is called language, as in language=pal.

```
t = fonttest(extras = {
    'specimen' : crossfont( text = u'Lorem Ipsum - Unicode pangram - Iñtërnâtiônàlizætiøn ' ↩
        , name = "Specimen", size = 10, featstr = "liga"),
    })
```

4.7.3 waterfall

This class creates a PDF file for each font and technology consisting of a waterfall of a string at various point sizes. It takes the following parameters:

text
The text string to be rendered in the font. If not included, it is taken from the global TESTSTRING, and failing that, is empty.

waterfalldir
Subdirectory under the test results directory into which to put the .tex and resulting .pdf files. If not present, uses the global WATERFALLDIR.

waterfallsizes
This is a number that specifies the font size in points. Defaults to the global TESTFONTSIZE (in points) and failing that to 12pt.

sizefactor
This specifies a factor to multiply each size by to get the linespacing. It default to the global TESTLINESPACINGFACTOR and defaults to 1.2

featstr
Specifies the feature string that should passed to XeTeX after the script information. Each feature is separated from the next by a colon. The language feature is called language, as in language=pal.

4.7.4 svg

This class is used for the generation of comparative SVG html reports to compare ot and graphite renderings. It takes the following parameters:

files
This dictionary gives all the test files to use and the feature values to use for rendering the text. The values follow the format used in the tex() class.

html
The html file that should be output that contains the links to all the test results.

diff
Specifies whether difference files should be generated. This value is a boolean.

4.7.5 tests

This class is used for managing regression type tests. In fact it can be used for any kinds of tests, since it allows the user to specify whatever tests they want and the commands involved. The first parameter passed to a tests() class contains a dictionary of test types (subdirectory names under the testsdir), and the cmd() to execute for that test type. For example, the default test list if none is specified, is equivalent to:

```
{ 'regression' : cmd('cmptxtrender -k -e ${shaper} -s "${script}" -t ${SRC[1]}
                      -o ${TGT} ${fileinfo} ${fileinfo} ${SRC[0]} ${SRC[2]}', shell=1) }
```

There are also some keywords that get expanded in the command string:

lang
The language tag for the test, whether extracted from the first few characters of the test file or specified as a parameter on the file in the file list.

script
The script tag. This is either specified in the files dictionary or comes from the script parameter in the font being tested.

shaper
Is either `ot` or `gr`, specifying which type of shaping is to be done in the test.

fileinfo
This is the value of the `extra` key in the `files` parameter dictionary for the particular file being processed.

In addition, a tests() takes the following named parameters:

standards
This directory is where to find the base fonts against which regression comparisons take place

files
As per the tex() files parameter.

ext
The default file extension for the `${TGT}` is `.log`, this can be changed using this parameter. For example: `ext='.pdf'`

shapemap
This is a function that is called with the `shaper` expansion and the result is used for that expansion instead.

coverage
This may take the following values: `fonts` indicates that the test is run once per font only. `shapers` indicates that the test is run once per shaper per font. In these cases there is no text file in the inputs.

5 Keyboards

Keyboard objects (`kbd()`) are much simpler than fonts. There are a number of attributes associated with a keyboard and they are listed here along with their default values:

source
The source file is a Keyman keyboard source file with extension `.kmn`. (Required)

target
The target file is the compiled Keyman keyboard and defaults to the source `.kmn` file with kmn replaced as `.kmx`

The following are attributes specific to generating a sample layout of the keyboard:

font
Font pathname to use for rendering the layout. This attribute is required if you want to generate keyboard layout documentation. It results in the font being copied into the build tree for this project.

fontname
Public font name that is used by the OS to reference the font.

fontdir
Directory containing the font specified in `font`. This defaults to `kbdfonts`.

kbdfont
Full pathname to the local font file copy to use for rendering the layout. By default this is built from the `font` and `fontdir` attributes. This attribute is rarely set. Use `font` instead.

fontsize
Font size to render glyphs at in the layout, given in pt (18).

modifiers
A list of strings which can take a sequence of the following strings, for example *Ctrl RAlt*

”
The empty string to indicate no modifier

Ctrl
Can be modified by L or R as well, as in LCtrl, RCtrl, Ctrl

Alt
Can be further modified by L or R or Gr, as in Alt, LAlt, RAlt, AltGr (equivalent to RAlt)

key
This specifies a key that is to be considered pressed before the keytop. This is used to document keyboard layouts following a deadkey. For example *[K_BKSLASH]* or \\ which are equivalent.

pdf
Filename to put keyboard layout pdf in (*source*.pdf)

svg
Intermediate svg format of keyboard layout (*source*.svg)

xml
Intermediate xml that represents the basic keystrokes of the keyboard (*source*.xml)

mskbd
An object that describes how to create an MS keyboard from the .kmn file.

An example specification of a keyboard object might be:

```
kbd(source = 'mykbd.kmn',
    fontname = 'My Font',
    font = 'results/myfont.ttf')
```

5.1 mskbd

The mskbd object can be used for the mskbd parameter of a keyboard. It automatically generates a normal MS Keyboard from the .kmn file and compiles it. The dependencies are pretty stiff, but it does mean that a complete MS solution can be generated without needing to run special GUI software during the development cycle. The parameters to mskbd() are:

lid
Language LCID to store this keyboard under. If not specified, defaults to 0xC00.

lidinstall
Specified as a list of language ids to install this keyboard under. E.g. lidinstall=[0x0436].

guid
GUID to associate with the keyboard. If not specified a random one is created.

capslockkeys
A list of keytops that should have capslock enabled for them. That is capslock will have a shift action (and shift will unshift). The keytop characters are: abcdefghijklmnopqrstuvwxyz0123456789-=[];’,./\`

The following are internal attributes that may be set:

arches
List of architectures that may be built for. Currently i586 and amd64. This is unlikely to be set by a user.

source
.kmn source file, defaults to keyboard .source attribute

c_file
.c file to generate (same as source but with .c extension)

rc_file
.rc file to generate (same as source but with .rc extension)

o_file
.o file to generate from .c file (same as source but with .o extension)

dll
.dll file to create (same as source but with .dll extension)

5.2 Tests

There are currently no automated tests for keyboards, although the generation of the layout can be considered a test.

6 Packages

Once a set of writing system components have been created, there is the need to package them for distribution. smith works to make that as simple but as powerful as appropriate. Each writing system component (`font`, `kbd`) has an optional `package` attribute. If this attribute is set, it is set to a package object corresponding to which package the component should be added to. In addition a global package object is created into which go all the components for which no `package` attribute has been set.

The global package can take its parameters from the wscript file as global variables with the same name as the attribute, but with the name uppercased.

The attributes to package are:

appname
Base application name to use for the installer in lower case. This is required.

version
Version number of the installer. This is required.

desc_short
One line description of the package.

desc_long
Multi-line description of the package.

desc_name
Multi-case name to use for the Windows installer generated.

outdir
Where to store the generated Windows installer relative to the build directory.

docdir
Directory tree to walk pulling in all the files as source files. Used for identifying documentation files. Also used in adding documentation files to installers.

license
License file to use within the package.

reservedofl
If set, uses the OFL and assembles a list of reserved font names from the constituent fonts for use within it.

zipfile
Name of zip file to use when creating smith zip. Is auto-generated if not set, based on appname and version.

zipdir
Directory to store generated zip file in, relative to build directory.

debpkg
debian package name for this package.

7 Tools

Here we install a number of pre-requisite programs that we will need later on. Unless otherwise stated, installers are assumed to take their default installation properties. With care you can change them if you want.

7.1 Ubuntu

Before installing packages, you should have `http://packages.sil.org/ubuntu` added as software source. Needed packages are:

- fontforge
- python-fontforge
- texlive-xetex
- grcompiler
- graphite-utilities
- nsis
- mercurial
- subversion
- libalgorithm-diff-perl

Optional but very useful packages include:

- meld
- kdiff3
- tortoisehg
- tortoisehg-nautilus
- subversion-tools

The following packages are needed, and should already be installed on an Ubuntu system.

- perl
- python

7.2 Windows

You will need to install the following programs:

- FontForge
- XeTeX - Paratext 7.x comes with XeTeX.
- Graphite compiler
- Graphite utilities
- NSIS
- TortoiseHg
- TortoiseSVN
- ActivePerl - Comes with Lib-AlgorithmDiff.
- ActivePython

7.3 Perl Modules

Many of the tools that smith uses are perl scripts. There are several ways to install these tools. You can unpack the sources anywhere on your system. If you need a suggestion of where to do this unpacking, please read the section *Directory Structure*.

7.3.1 Packages

If you are running Ubuntu, installed the following packages

- libfont-ttf-scripts-perl
- libfont-ttf-perl

If you are running Windows you can get tarballs (`.tar.gz`) from CPAN for the following packages.

- Font-TTF-0.46.tar.gz
- Text-PDF-0.30.tar.gz
- Font-TTF-Scripts-0.13.tar.gz

The versions will probably be newer by now. Extract the tarballs and follow the steps for installing from subversion repositories. You can use a program such a `7-zip` to extract the tarballs.

7.3.2 Subversion Repositories

Here we install various Perl modules from source. The reason for doing this is so that we can update any software that we might encounter problems with. The basic procedure for installing a perl module is:

- run `perl Makefile.PL` to create a Makefile
- run `make` to create the local installation copy
- run `make install` to install the local copy to the main perl installation

First you need to create three local repositories. You will only need to do this once. Run the commands

```
svn co http://scripts.sil.org/svn-public/utilities/Font-TTF/trunk Font-TTF
svn co http://scripts.sil.org/svn-public/utilities/Text-PDF/trunk Text-PDF
svn co http://scripts.sil.org/svn-public/utilities/Font-TTF-scripts/trunk Font-TTF-scripts
```

Every time you wish to update and install the perl tools run the following commands. On Windows you will need to replace `make` with `make DFSEP=/` due to the vagueries of GNU make. You should also change the command `sudo make install` to `make install`.

To install Font-TTF:

```
cd Font-TTF
perl Makefile.PL
make
sudo make install
cd ..
```

To install Text-PDF:

```
cd Text-PDF
perl Makefile.PL
make
sudo make install
cd ..
```

To install Font-TTF-Scripts:

```
cd Font-TTF-scripts
perl Makefile.PL
make
sudo make install
cd ..
```