Prescript: Programme Structure and Functional Description

David J Miller (djm6)

March 4, 1998

1 Introduction

Prescript is a tool to convert postscript source into raw ASCII text or formatted HTML. When outputting in HTML, it is designed to preserve as much of the page format as it can by predicting the linebreak that terminates each line.

Using the prediction, the page can be marked up in such a way that the meaning of each line termination is preserved. So for example, if you are looking at the HTML version of a paper in a web browser, the paragraphs will wrap to fit the window width (which is what should happen with paragraphs), rather than literally reproduce the original page format.

The purpose of this document is to describe the over all structure and design philosophy, and also explain some of the strangenesses in it. It is neither the purpose of this document to explain how each line of code works, nor what each function and class method does exactly. These things should be sufficiently evident by reading the source, as I have tried to be fairly liberal with comments. Most functions have descriptions, as do most important variables.

In this document, I have tried to pre-empt the kinds of things you might want to know about how *Prescript* was written. This is a difficult task at best, especially when it comes to explaining why certain things are the way they are when the answer is, "for no reason other than the historical development of the *Prescript* ." This document attempts to describe things at a progressively more detailed level without duplicating the documentation contained within the code comments.

2 Programme Structure

2.1 Module Organisation

There are six modules as follows:

prescript The main module, which co-ordinates all operations.

io.py Routines responsible for input and output. It incorporates code to generate and read the text fragments from the postscript source, and routines for rendering and doing the output.

process.py All intermediate processing occurs here. This includes line assembly, page assembly and document assembly. Almost all the data gathering and processing occurs here with the exception of the prediction itself. The reasons for this are largely historical, but also having the algorithm coded in a separate module makes it very clear how it works and what the algorithm rules are.

predict.py This contains the code implementing the algorithm that predicts the kind of line termination for each line. Although the algorithm is coded in a very straight forward and easy to understand manner, it may not be immediately obvious how we came to some of these rules. Some of them are observation, some of them are trial and error.

We accept that it is not possible to obtain 100% accuracy for the huge variety of document formats and layouts that exist. So for this reason, we have attempted to formulate rules that produce the least errors most often.

misc.py General functions etc that don't fit anywhere else go in here. Many of the functions that are in there were written for previous versions of *Prescript* and are in there for compatibility.

statfunctions.py Statistical functions.

2.1.1 prescript

Everything is fairly straight forward and self-evident in here. There are only two things worthy of note:

`prescript_dir` is a global variable which points to where *Prescript* 's executables (esp, prescript.ps) are.

`format` is a global variable which says what the output format is.

2.1.2 io.py

This module contains:

Classes

Those related to input are:

`Fragment` Represents a single text fragment. It is able to concatenate another fragment onto itself, and it is able to tell you the average character width.

`PSDatReader` An instance of this class is passed to `readPostScriptDataFile`. It contains methods to deal with fragments and pagebreaks, and acts as a container for these items of data. It also contains code for generating one statistic that is needed by the next pass, `modeCharWidth`.

Those related to output are:

`PlainTextFormatter` A formatter class which implements raw ASCII output with minimal formatting.

`ARFFFormatter` A formatter class which implements ARFF output. It only outputs plain lines (ie, not page numbers and not headers or footers.) See implementation for details.

`HTMLFormatter` A formatter class which implements incomplete HTML output. It is incomplete because it does not output a header or the `<HTML>` tags, only what appears between `<BODY> .. </BODY>`

Variables

`knownFormats` A list of strings naming the output formatters that are implemented in the module. Currently, supported formats are: *plain, html, and arff.*

`maxBadLines` The maximum number of errors that may occur during the reading of fragments before *Prescript* bails. There should never be *any* errors, except of course when the input file is not postscript in which case the text *Prescript* is trying to process is infact the error output from ghostscript.

Functions

Those related to input are:

`readPostScriptDataFile` Provides the mechanism for reading fragments from the given file. This function deals with bad input fragments, and of those that are valid it does the appropriate action based on the kind of fragment read. It also takes care of weird recursive calls which cause irregular fragments to be produced.

`readFragments` Generates the fragment data by invoking ghostscript on the given filename. Returns the instance of PSDatReader, full of fragments.

`applyHandcheck` Reads and applies the hand-checked classification file associated with `inputFilename`.

Those related to output are:

`renderPage` Renders the given instance of Page using the given formatter.

`renderDocument` Renders the given instance of Document using the given formatter.

`newFormatter` Creates and returns an instance of the appropriate formatter class based on the string content of `format`

2.1.3 process.py

This module contains:

Classes

`Line` The humble line. Knows how to set all its own numerical attributes. After the fragment, it is the fundamental unit.

`Page` A collection of `Line` instances, with a few extra methods for calculating statistics required for setting other attributes in each `Line`. This is a rather complex and involved class, probably more so than it should be.

`Document` A collection of `Page` instances, with methods capable of looking after page order and generation of yet more very important statistics required to set the rest of the attributes in each `Line`.

The remaining classes in this module don't represent any particular thing, instead they represent units of work or processing to do. They serve to keep function boundaries clear, and also container classes for the data in intermediate form. In general, only one instance of each of these classes is needed during the entire programme.

`LineAssembler` Responsible for assembling a line of text from the fragments submitted to it. If you suspect there is a problem with the way fragments are joined, this is the culprit.

For each line of input read, a new instance of `Line` is created which is subsequently submitted to a page assembler.

`PageAssembler` Responsible for assembling whole pages. Appart from calling the appropriate methods of a document assembler at the appropriate times, this class does little.

`DocumentAssembler` Same as idea as `PageAssembler`.

Variables

All the important variables appear at the top of this file with plenty of description. I won't duplicate it here.

Functions

`assembleDocument` Takes a collection of fragments in the form of a `PSDatReader` instance and returns an instance of `Document`.

`preprocessDocument` Causes the remaining unset attributes to be set in each line.

2.1.4 predict.py

This module contains:

Functions

`predict` Predicts the linebreak for the given instance of `Line`.

`predictDocument` Invokes the predict on each line in the given `Document` instance. Because it is convenient to do so, it also invokes the method Page.dehyphenateLine

2.2 How Things Fit Together

Python is a slow language. It's got a nice syntax, but it's dog slow, and people have complained about this. Consequently, I have attempted to balance program structure against program performance.

There are some instances where I have sacrificed good program design for good performance. An example of this is `PSDatReader`. The name of the class implies that its function is to produce a data structure containing the text fragments, but it does not suggest that it collects statistics as well.

It might be clearer to code these two steps separately, but to do so is a waste of a pass, so I have packed as much work into each pass as possible.

In addition, a number of changes were made right near the end, and they weren't properly thought through. In any case, if you have suggestions about how the program could be coded better, I would be grateful to know what they are.

2.2.1 Summary

- Fragments are generated and put into a container *(module io)*.
- Lines are assembled, pages are assembled and then document is assembled *(module process)*.
- Attributes, parameters, etc are calculated (including AVLS, MOLS) *(module process)*.
- Prediction is made, and dehyphenation is done *(module predict)*.
- If appropriate, the handcheck file is applied.
- Document is rendered and output.

2.2.2 A longer description

After the main module initialises itself, it calls `io.readFragments` which calls ghostscript and puts everything into an instance of `PSDatReader`. `PSDatReader` functions as a container class, and it also gathers the document-average character width.

This statistic is needed for the next pass to determine whether the next fragment is a part of the same word, a part of a new word or the start of a new line.

For each fragment read from ghostscript, `PSDatReader.textFragment()` is invoked. For each pagebreak encountered, `PSDatReader.newPage()` is invoked. Finally, `PSDatReader.done()` is called.

Having returned from `io.readFragments`, the next step is to assemble lines, pages and finally, the document. This task is done by a single instance each of `LineAssembler`, `PageAssembler`, and `DocumentAssembler`.

In order to create an instance of `LineAssembler`, an instance of `PageAssembler` is required, and in order to create an instance of that, an instance of `DocumentAssembler` is required.

For each fragment, `LineAssembler.submit()` is called. For each line assembled, `PageAssembler.submit()` is called. For each page assembled, `DocumentAssembler.submit()` is called. Each assembler has a `done` method, and some of them have a `newPage()` method, each of which is called at the appropriate time. The done methods are used to gather statistics etc

The document is now fully assembled, but not all of the attributes have been calculated. Many of the attributes depend on the margins of the page or the mode values for linespace and linelength. All these are calculated with a call to `Document.preprocess()`. What goes on in there gets messy. The best thing to do is look at the source for details. A summary of what happens follows:

- Linespace "polarity" is checked. Some documents put the origin at the bottom of the page instead of the top, so the linespace values are all negative. This is a problem. The checking process determines if the sign of the linespace values need to be reversed and does so if necessary.

- Mode linelength and mode linespace values are calculated over the entire document. These are two very important metrics.

- Column margins are calculated on a page by page basis. This has the effect of determining the number of columns in each page. The document-wide column margins are calculated by taking the mode column margins.

  In fact, only the right margin is calculated. There are problems with calculating the left column margin that outweigh the benefits of having such information.

Having determined these parameters, `Document.preprocess()` calls the `calcValues()` method of each page in the document, which sets the remaining attributes of each line.

By now, all calculations and all data gathering has been done, and the document is now ready to be submitted for prediction, which occurs in the `predict` module. This module is little more than the algorithm in valid python syntax. It should be very straight forward and obvious what is going on there. This is the place where modifications to the algorithm should be done.

The output stage is also relatively straight forward. The document is applied to the formatter by calling the formatter's `line` method for each line and `pagebreak()` method for each pagebreak. In addition to that, the prediction is called *as a method*. For example, for *explicitlinefeeds*, the method `formatter.explicitlinefeed()` is invoked.

This means that if you change in any way output of the algorithm, you must make sure there are the appropriate methods in the formatter to cope with the changes.

This mechanism makes adding new formats trivial. See the section on modifying *Prescript* for details.

3 Esoterics

Equalness There are several so-called *equalness* parameters that specify how equal is equal. Because of the nature of the programs that produce postscript files, there can be a slight variation in values which for the purposes of comparing equality, are not important and should still be considered "equal". The equalness paramaters define the slackness when comparing equality with the function `misc.isEqual()`.

ARFF Output There is a hard-coded file 'arff-header' which may exist in PRESCRIPT_DIR. This will be prepended to the ARFF file output.

4 Modifying *Prescript*

4.1 Adding New Formats

The notion of formatter classes make implementation of new formats very straight forward. All these changes are done in the `io` module.

1. Create a new formatter class that implements the `linefeed()`, `paragraph()`, `explicitlinefeed()`, `pagebreaklinefeed()`, `pagebreakparagraph()` methods. In addition, there are two methods start(), and end() which are called before and after the document is submitted to allow such things as headers to be output.

2. Add the new format into the `knownFormats` variable

3. Add support into `newFormatter` function.

4.2 Modifying Attributes Output into ARFF Files

The definition of the `Line` class is in the `process` module. Python classes may have a method called `__str__()` which defines the string representation of that class. This is what is used when outputing lines in ARFF format.

If you wish to change the number or order of attributes output in ARFF files, modify the tuple "`output`" located in this method to reflect the order and number of attributes output. Each attribute should start with "`self.`" and be found in the class initialiser (`__init__()`).

4.3 Modifying Prediction Algorithm

This is fairly straight forward also. Modify the function `predict` in the `predict` module to suit. You may use any attribute found in the initialiser of the line class initialiser (`Line.__init__()`), but it is preferable that only boolean attributes are used.

Once again I stress, if you modify the symbols output by the prediction algorithm, make sure you modify ALL of the formatter classes in module `io`.

4.4 Adding Attributes

This is the procedure for adding new attributes:

1. Add the attribute name to (`Line.__init__()`), remember to prepend "`self.`". In python, the scope rules assume references to variables that are not resolved to being in a particular object, (including self) to be local, unlike C++. Make sure you provide an initial value. The preference for boolean attributes is `NO` and for numeric attributes `'?'`

2. Add the new attribute into `Line.__str__()` if desired.

3. Modify the appropriate part of the code in module `process` to set the attribute to its correct value. The appropriate place is likely to be `Page.calcValues`.

4. Modify the algorithm in module `predict` as appropriate.