NEA Report

Christopher Lang 1132

Typesetting System

2022-2023

Contents

1 Analysis

1.1 Problem Statement

To create a document preparation system for project reports.

1.2 Problem Background and Analysis

As a student, I often need to create digital, printable documents. I have experimented with a range of document preparation systems: Microsoft Word, Markdown, LaTeX, etc. But I am yet to find a solution that is simultaneously simple, powerful and easy to use. This project intends to achieve all three of these goals.

A major part of document preparation is text processing. This involves converting markup into formatted lines of text. Markup describes the content and can influence the formatting of the text. Digital text processing systems typically come in three types * : presentational, procedural and descriptive.

Presentational systems provide a GUI "what you see is what you get" (WSYIWYG) editor to enable the user to modify the documents markup. Text is typeset and displayed to the user as they write. This type of editor is often easy to use, but it is difficult to have a wide range of capabilities within the confines of a GUI/window system [1].

Procedural markup consists of a sequence of commands that instruct the software how to format the text. By combining a small set of simple commands, complex behaviour can be achieved. However, as commands are processed sequentially, it is often difficult to have the formatting of earlier text depend on later content. Consider a two-column page with footnotes that span the entire page. The first column is written and formatted with no footnotes and then a footnote appears on the second column which reduces the height available for the first column. Clearly, the first column must be re-formatted to make it's text shorter. But what if this causes the footnote to appear on the next page - now the first page's columns are shorter for no reason! As you can see, procedural markup has difficulty in solving certain typesetting problems.

Descriptive markup identifies what each part of text IS and not how to typeset it. For example, a header may be surrounded in *<header>* *</header>* tags. Then, the text-processing system is responsible for deciding how to format this header. This system is more flexible than procedural markup, the footnote problem described in the last problem can be solved simply by marking a section of text as a footnote and relying on the typesetting software to insert it in the right page. However, 'tags' cant be combined as effectively as Procedural markup commands, this means descriptive markup languages can quickly become very complicated with a huge number of 'tags' that the user must know (think HTML).

Apart from text-processing, document preparation systems must be able to insert graphics into page content and output to a printable file format. PDF and PostScript

* *Coombs, James H.; Renear, Allen H.; DeRose, Steven J. (November 1987). Markup systems and the future of scholarly text processing. Communications of the ACM 30 http://xml.coverpages.org/coombs.html*

are the most common of such formats. PostScript is the older of the two formats, it is a text based format which may make it easier to generate. PDF is more widely used, has a richer set of features and its standard is better documented.

1.3 Intended End-User

Users of Unix-like operating systems who need to generate PDF documents.

1.4 Third Party

Anthony Ceponis uses a Linux based operating system and has recently finished writing a computer science NEA using 'google docs' - a WYSIWYG editor. The following is an extract of the transcript of a conversation I had with Anthony in order to better understand the requirements of the end-user.

Christopher: Whats your general opinion of writing in a WSYIWYG editor compared to writing documents in a markup language like HTML?

Anthony: With a GUI editor, everything is much easier and convenient and quicker (unless your WPM is out of this world) and more accessible to the average person.

Christopher: Did you include a copy of your source code in the NEA report?

Anthony I was writing the NEA documentation using google docs and I had to endure the painful process of taking a screen shot of over 10,000 lines of code in small chunks and copy pasting them into my report as part of my technical solution.

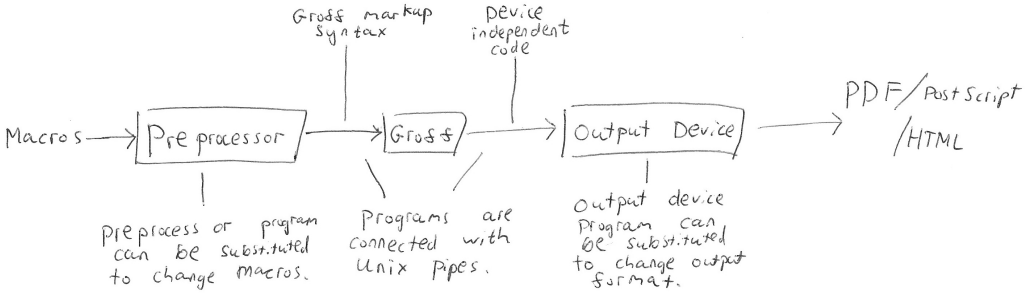
Christopher: Would you benefit from a program that was capable of automatically inserting the source code into the PDF document?

Anthony I would do unspeakable thinks to have access to a program/feature that would achieve what you just described.

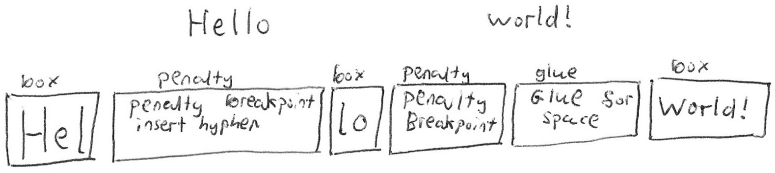
1.5 Research and Modelling

To embark my research, I examined some existing solutions:

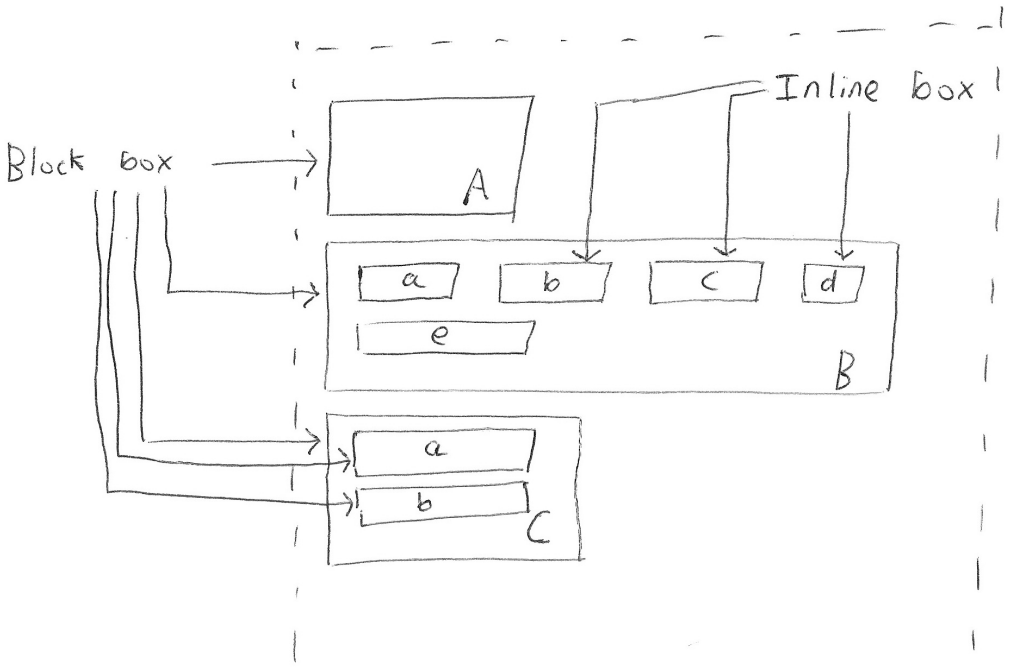
groff. First released in 1990, groff is GNU's replacement for troff. Like troff, groff makes use of Unix pipes to process documents in several modular stages. Groffs compatibility with Unix systems is, in my eyes, its greatest strength because, unlike graphical editors, Groff can be invoked programmatically to take input from existing files or streams. In addition, the groff input file syntax and syntax of many groff preprocessors is designed to be simple to parse and generate through Unix streams. This makes it easy to write programs to process or modify the document before it is typeset. The groff manual [1] was a valuable resource in understanding how procedural markup languages can solve difficult typesetting problems in a single pass. The following diagram shows how Unix pipes are used to modularize the typesetting process. Each box is a binary program and each arrow is a Unix pipe.



TeX + LaTeX. TeX is a typesetting system first released in 1978. I used a derivative of TeX, LaTeX, to write a number of documents for a school project. My frustration with LaTeX's unreadable syntax and confusing extension system was the initial impetus to make an alternative. The TeXbook [2] discusses a model of untypeset content consisting of an ordered list of *gizmos* each representing an atom of content. This model was an promising starting point in my thoughts about how to best define document content before it is typeset. Donald E Knuth's paper 'Breaking Paragraphs into Lines' [3] describes the line breaking algorithm used by TeX and compares it to the more primitive 'first fit' method. The following image shows how *box, glue,* and *penalty* gizmos are used to model the text "Hello World!".



HTML + CSS. Although browsers do not need to split webpage content into pages, text and graphics must still be arranged on the screen depending on resolution. I read an online article about how browsers work [4] and a blog about building a browser engine [5] to understand this layout process. The hierarchical DOM provides an alternative document content model to TeX's linear *gizmos.* The following diagram shows how CSS "block" and "inline" boxes are placed on a page.



In order to better understand the technical requirements of the project, I conducted some more specific research into the implantation details.

TrueType Reference Manual. [6] This reference manual defines the binary file format of 'true type' font files. Parsing a font file to extract glyph widths and other such data is an essential step in the typesetting process.

ISO 32000-1 Portable Document Format 1.7. [7] This document is the technical specification for the PDF 1.7 file format. The project intends to generate PDF files so understanding the file format is necessary.

Single-source shortest path in DAGs. [8] This single-source shortest path in directed acyclic graphs algorithm can be used to solve the line breaking problem as modeled in Donald E. Knuth's 'Breaking paragraphs into lines' [3] in O(V+E) time. In practice, not all feasible lines are known before the algorithm starts, so the algorithm will need to be modified to search for the feasible edges as it progresses through the text.

1.6 Objectives

I have decided to make document preparation software which uses a text-processing system of the 'descriptive' type (though some procedural features may be supported). After considering my online secondary research and discussion with Anthony, I have produced the following set of specific and measurable objectives.

```
Typeset PDF Document
  1. Parsing Input
    1.1 Escape sequence identifies bold text
    1.2 Escape sequence identifies italic text
    1.3 Escape sequence identifies header text of multiple sizes
    1.4 Escape sequence identifies footnote
  2. Break text into lines
    2.1 Optimal line breaks are selected to minimize total trailing whitespace
```

```
  2.2 Line breaks can insert text when a break does not occur (for example insert a space)
  2.3 Line breaks can insert text at the end of a line it breaks (for example a hyphen)
  2.4 Text of varied fonts and sizes can be mixed in the same paragraph
3. Break lines into pages
  3.1 Lines are fitted onto pages to minimise empty spece at the end of each page
  3.2 Footnotes are inserted at the bottom of the page
  3.3 A footnote must appear on the same page it's referenced
  3.4 Images can be inserted into the content
  3.5 Page numbers can optionally appear at the bottom of each page
  3.6 A contents page can be added which automatically locates relevant page numbers
  3.7 User-specified header text will appear at the top of each page
  3.8 Margin sizes can be controlled by the user
  3.9 Pages are written to a PDF file
  3.10 Fonts used are embedded into the PDF file
  3.11 Images used are embedded into the PDF file
```

1.7 Prototype

In order to demonstrate the feasibility of the project, I wrote a python script to generate a simple PDF file. After successfully writing a one-page PDF file containing text of a built-in font, I decided that the project was likely achievable.

2 Documented Design

2.1 Operating System

This project is intended to work on Unix-like machines only because it requires use of Unix pipes.

2.2 File Structure and Source Control

git was used for the projects source control combined with *GitHub* to backup the repository in the cloud. A secondary branch was created for the excessive code commenting and report required by the NEA specification.

Source files are stored in the root 'typewriter' directory. I did not feel the need to place them in a nested folder structure as they are relate more procedurally than hierarchically. In addition, it is easier and faster to access them when they are all in the root folder. Object files and binaries are also built into the same root directory.

The 'report' directory contains, source files and image for generating this report along with symbolic links that point to the sample 'typeface' file and 'fonts' directory located in 'typewriter'. These symbolic links are read by 'report.sh' when generating this report. The 'test' directory contains test documents and similar symbolic links.

```
typewriter
|-- contents.py
|-- dbuffer.c
|-- demo
|   |-- demo.mkv
|   |-- fonts -> ../fonts
|   |-- objective_1-parsing_input.sh
|   |-- objective_2-break_text_into_lines.sh
|   |-- objective_3-break_lines_into_pages.sh
|   |-- output.pdf
|   |-- peppers.jpg
|   `-- typeface -> ../typeface
|-- fonts
|   |-- bybsy.ttf
|   |-- cmu.serif-bold.ttf
|   |-- cmu.serif-italic.ttf
|   |-- cmu.serif-roman.ttf
|   `-- cmu.typewriter-text-regular.ttf
|-- jpeg.c
|-- LICENSE
|-- line_break
|-- line_break.c
|-- Makefile
|-- markup_raw.py
|-- markup_text.py
|-- markup_text.py.orig
|-- markup_text.py.rej
|-- pager.py
|-- pdf.c
|-- README
```

```
|-- record.c
|-- report
|   |-- css.jpg
|   |-- dag.jpg
|   |-- demo_content.sh
|   |-- fonts -> ../fonts
|   |-- gizmos.jpg
|   |-- groff.jpg
|   |-- nea.pdf
|   |-- peppers.jpg
|   |-- report.sh
|   |-- state_machine.jpg
|   |-- typeface -> ../typeface
|   `-- upload.sh
|-- test
|   |-- doc.sh
|   |-- fonts -> ../fonts
|   |-- maths.sh
|   |-- output.pdf
|   |-- peppers.jpg
|   `-- typeface -> ../typeface
|-- ttf.c
|-- tw
|-- tw.c
|-- tw.h
|-- typeface
|-- utils.c
`-- utils.py

8 directories, 50 files
```

2.3 Executable Files

In order to modularise the typesetting process and make the project more compatible with Unix systems, multiple independent executable files are built each with responsibility of part of the document preparation process. These executables communicate with each other through Unix pipes using these well defined file formats: *'text specification', 'content', 'pages', 'contents'.*

tw. tw (short for typewriter) is the program at the core of the project and is often the last step of the typesetting process. It converts the stdin stream of the *pages* format to a PDF file. The '-o' option can be used to specify where to output the PDF file. The 'pages' format defines what graphic elements are to appear on each page and where. tw must embed fonts and images into the PDF file, as well as writing the text content. The value of this program is in the abstraction it provides over the complex PDF file format. It is much easier to write a program to generate the *pages* format and then to pipe that to tw than to write a program which generates PDF files directly.

pager. This executable reads the *content* format from stdin, splits this content into *pages* which are written to stdout. *contents* is also written to a file. This contains a table of 'marks' (a component of the *content* format) and the pages which these marks are located. The *contents* file is essential in implementing a contents page. Page breaks may only occur at optional breakpoints specified by the *content* format. Some specified points in *content* must be located on a new page. pager must consider footnotes and allocate sufficient space at the bottom of the page for them to fit. When both regular content and footnote content is added in between optional breaks, both types of content must be located on the same page. Command line options are provided which control page margins, page numbers and where to output the contents file.

contents. This program converts the *contents* table read from stdin into *content* which is written to stdout. Each entry in the table is converted into a line of text starting with the mark name, ending with the page number and padded with dots to achieve a

set character width. A monospaced font is used so that the constant character width translates into a constant line width. Command line options can be used to set the character width and the font size.

line_break. This program reads *text specification* from stdin, calculates the optimal line breaks and writes *content* to stdout. The *text specification* defines what text is to appear, with what font and size, where line breaks can occur and where line breaks must occur. Optional breaks can insert text depending on weather the break occurs. For example, an optional break between two words inserts a space when no break occurs and an optional break within a word inserts a hyphen when a break occurs. Line breaks are chosen to minimize the sum of surplus white space on each line. The line break algorithm considers the entire paragraph when doing this; the last word in a paragraph can affect the first line break. Command line options can be used to set text align mode and line width. Left, right, centre and justified align modes are supported.

markup_text and **markup_raw.** The markup programs read a human readable markup language from stdin and write *content* to stdout. markup_raw writes the text 'as is', maintaining line breaks. Font, maximum orphan lines and maximum widow lines can be set with a command line options. Orphans are lines of text that appear alone on the bottom of a page, disconnected from the rest of their body of text. Widows are isolated lines of text, alone at the top of a page. markup_text invokes line_break to calculate the optimal line breaks for the text it is given. Text enclosed in stars (*) and underscores (_) is interpreted as bold and italic text respectively. Lines beginning with hashtags (#) are headers which change in font size depending on the number of hashtags at the start of the line. Lines beginning with a carrot symbol (^) are footnotes, the remainder of text in the line is the content of the footnote associated with the last word before the footnote. Command line options are provided to control text width, size, align mode, line spacing and paragraph spacing for both footnotes text and normal text.

2.4 File Formats

2.4.1 Records

CSV files seem to be the standard way of encoding a table of strings in a text file. However, when fields include text that may contain escaped commas, it can be confusing to read and appear cluttered. For my project, I have developed an alternative format which encodes a variable number of string fields on each line. This *'records'* format is used frequently in the projects text streams.

Lines are separated by a single new line character (LF not CRLF). Empty lines and lines consisting of only spaces are ignored. Spaces at the beginning and end of lines are ignored. Fields are separated by one or more spaces. A field consists of EITHER a sequence of characters none of which are spaces or new lines OR a sequence of non new line characters enclosed in double quotation marks ("). In each type of field, a backslash can be used to escape the next character. For example a backslash followed by a space (\ ) represents a space literal that does not begin the next field and a backslash followed by a double quotation mark (\") represents a double quotation mark that does not end the field. A double backslash sequence (\\) encodes a backslash literal. Fields beginning with a double quotation mark must be closed with another double quotation mark before the end of the line.

```
this line has 5 fields
this line has 4\ fields
"this line has 1 field"
field_0 "field 1" "field \"2\"." field\ 3
```

2.4.2 Pages

There are three modes in the *pages* format: document, graphic, text. Each mode

interprets records differently. The first line of a *pages* file is interpreted in document mode.

Document Mode Commands:

START PAGE is the only valid document mode command. It adds a new page to the document and enters graphic mode. The graphic read in this graphic mode will define the page content with origin (0, 0). Parsing will return to document mode when graphic mode is exited. When back in document mode, subsequent pages can be added with more 'START PAGE' records.

Graphic Mode Commands:

MOVE [x_offset] [y_offset] This graphic command sets the *current position* to the graphic origin plus (x_offset, y_offset).

IMAGE [width] [height] [jpeg_file_name] Draws a baseline DCT-based JPEG image on the page at *current position* with width and height measured in points.

START GRAPHIC Enters a new graphic mode with origin *current position.* This graphic is drawn on the same page.

START TEXT Enters text mode. The text read in text mode is painted on the page at *current position.*

END Exits graphic mode.

Text Mode Commands:

FONT [font_name] [font_size] Sets the active font.

STRING [string] Must appear after the first FONT command in this text mode. Adds *string* with the active font to the text to be drawn.

SPACE [word_spacing] Sets the *word spacing* for subsequent STRING commands. The word spacing is the additional width to add to space characters measured in thousandths of points. This is used in text justification.

END Exits text mode.

The following is an example *pages* stream:

```
START PAGE
MOVE 100 100
IMAGE 400 400 peppers.jpg
MOVE 60 766
START GRAPHIC
MOVE 100 0
START TEXT
FONT Regular 36
STRING "Title"
END
END
MOVE 60 742
START TEXT
FONT Regular 12
STRING "hello"
STRING " "
STRING "world"
STRING " "
SPACE 10000
```

```
STRING "this"
STRING " "
STRING "is"
STRING " "
STRING "some"
STRING " "
STRING "text"
END
END
START PAGE
MOVE 100 300
START TEXT
FONT Regular 12
STRING "document end"
END
END
```

2.4.3 Content

Content is parsed by a *pager* program which splits the content into pages. A *content* stream consists of a sequence of content commands. Some content commands are followed by a *pages* graphic which must begin with a record with first field 'START'. This graphic continues until the number of records encountered with first field 'START' is equal to the number of records encountered with first field 'END'. The following is a list of content commands and their function.

flow [flow] Sets the current *flow*. This may either be 'normal' or 'footnote'.

box [height] The following graphic with *height* measured in points is to be added to the current page. The current *flow* determines weather the graphic is to be placed in the main body of content or in the footer. If the box does not fit in the current page then a new page is added.

glue [height] If the previous box appears on the same page then the next box must be vertically padded by *height* points.

opt_break A page break may occur at this location Every valid page break location must be explicitly defined.

new_page If any optional breaks already appear on the current page then the following content must appear on a new page.

Example *content* stream:

```
box 12
START TEXT
FONT Regular 12
STRING "Hello"
STRING " "
STRING "world,"
STRING " "
STRING "this"
STRING " "
STRING "is"
END
opt_break
box 12
START TEXT
FONT Regular 12
STRING "an"
```

```
STRING " "
STRING "example."
END
opt_break
```

2.4.4 Contents

The *contents* format defines information to be displayed in a contents page. It is generated by a *pager* and is the input to a *contents* program to generate contents page content. The format consists of a number of records. Each record must have 2 fields: the first is the name of some content to be marked in the contents page, the second is the page number in which this content appears. The page number does not need be an integer, for example if roman numerals are used for page numbers. The marks will appear in the contents page in the same order they are defined in the contents stream. The following is an example contents stream.

```
"Preface" iv
"Introduction" vi
"Section 1" 1
"  Section 1.1" 2
"  Section 1.2" 5
"Section 2" "7"
```

2.4.5 Text Specification

The text specification format defines text before it is broken into lines. The line_break program parses text specification and selects optimal line breaks. The format consists of a number of records, parsed sequentially. The first field in each record is a 'command name' that defines the meaning of subsequent fields in the record. The following text lists each supported command and its purpose.

FONT [font_name] [font_size] This record must consists of 3 fields. The first is the string 'FONT', to identify the command type. The second is a string that identifies the font name as defined in the *typeface* file. The third field must be an integer, the font size. This command sets the font to be used in subsequent STRING and OPTBREAK commands.

STRING [string] This command must appear after the first FONT command. The string with the most recently set font is appended to the list of gizmos which the line break algorithm will operate on.

OPTBREAK [no_break_string] [at_break_string] [line_spacing] Appends an optional break gizmo. If no break occurs here, the no_break_string with most recently set font is inserted into the current line in-place. If a break does occur here, the at_break_string is added onto the end of the complete line, also with the most recently set font. The new line will be vertically padded by line_spacing points from the previous line.

BREAK [line_spacing] A line break must occur here. The new line will be vertically padded by line_spacing points from the previous line.

MARK [identifier] For the line that contains this mark: an additional record is inserted into the content output inbetween the line's box graphic and the content optional break. This record consists of a carrot symbol (^) followed by the identifier. The mark command is used for locating the line with which a footnote that is associated with a word in that line must appear.

The following is an example of the *text specification* format.

```
FONT Regular 12
STRING "Hello"
OPTBREAK " " "" 0
STRING "world."
FONT Bold 12
OPTBREAK " " "" 0
STRING "bold"
OPTBREAK " " "" 0
STRING "text."
FONT Regular 12
OPTBREAK " " "" 0
STRING "hyphenat"
OPTBREAK "" "-" 0
STRING "ion."
BREAK 12
STRING "New"
OPTBREAK " " "" 0
STRING "paragraph*"
MARK cite_paragraph
OPTBREAK " " "" 0
STRING "with"
OPTBREAK " " "" 0
STRING "a"
OPTBREAK " " "" 0
STRING "footnote."
```

2.4.6 Typeface

The *typeface* format is found in the typeface file (which must be named 'typeface'). This file must be located in the current working directory when a program that needs it is run. Its purpose is to map a font name to a font file. Each record consists of two fields: the first is a font name, the second is a file path to the corresponding font file. The file path may be relative to the current working directory.

Example:

```
Regular /usr/share/fonts/cmu.serif-roman.ttf
Italic cmu.serif-italic.ttf
Bold cmu.serif-bold.ttf
Monospace cmu.typewriter-text-regular.ttf
```

2.5 Data Structures

2.5.1 Dynamic Buffer

```
struct dbuffer {
  int size, allocated, increment;
  char *data;
};
void dbuffer_init(struct dbuffer *buf, int initial, int increment);
void dbuffer_putc(struct dbuffer *buf, char c);
void dbuffer_printf(struct dbuffer *buf, const char *format, ...);
void dbuffer_free(struct dbuffer *buf);
```

This structure is usefull in string building. *data* points to a region of memory allocated on the heap *allocated* bytes in size. *size* measures how many bytes of *data* are being used. As data is added to the buffer, if more than *allocated* bytes are needed then more bytes are allocated in increments of *increment. dbuffer_printf* and *dbuffer_putc* functions append new characters to the dynamic buffer.

2.5.2 Record

```
struct record {
  struct dbuffer string;
  int field_count;
  int fields_allocated;
  const char **fields;
};
void init_record(struct record *record);
void begin_field(struct record *record);
int parse_record(FILE *file, struct record *record);
int find_field(const struct record *record, const char *field_str);
void free_record(struct record *record);
```

The record structure stores a variable number of strings which are referred to as 'fields'. *strings* is a dynamic buffer that contains each field separated and ended with null characters. *fields* is a pointer to a heap-allocated array of pointers to the first byte of each field in *strings. fields* has been allocated *allocated* bytes on the heap, this amount may increase as more fields are added. There are *field_ count* valid and unique entries in *fields.*

2.5.3 Font Info

```
struct font_info {
  int units_per_em;
  int x_min;
  int y_min;
  int x_max;
  int y_max;
  int long_hor_metrics_count;
  int cmap[256];
  int char_widths[256];
};
int read_ttf(FILE *file, struct font_info *info);
```

font_ info contains the data parsed from a true type font file. Character widths are measured by thousandths of points the character would occupy if drawn with font size 1.

2.5.4 Line Break Gizmos

```
struct gizmo {
  int type;
  struct gizmo *next;
  char _[];
};

struct text_gizmo {
  int type; /* GIZMO_TEXT */
  struct gizmo *next;
  int width;
  struct style style;
  char string[];
};

struct break_gizmo {
  int type; /* GIZMO_BREAK */
  struct gizmo *next;
  int force_break, total_penalty, spacing, selected;
  struct break_gizmo *best_source;
  struct style style;
  int no_break_width, at_break_width;
  char *no_break, *at_break;
```

```
  char strings[];
};

struct mark_gizmo {
  int type; /* GIZMO_MARK */
  struct gizmo *next;
  char string[];
};
```

The gizmo linked list is the data structure used in the line breaking algorithm. A *gizmo* is a variable-size region of heap-allocated memory. The first *sizeof(int)* bytes are used to identify the type of gizmo: text gizmo, break gizmo or mark gizmo. The next *sizeof(char *)* bytes are a pointer to the next gizmo in the list. The meaning of subsequent bytes is dependent on the type of the gizmo. Each gizmo type ends with a variable number of characters which may be used as a place to store null-terminated strings which can be pointed to in the gizmo. In the case of the *text_gizmo,* and *mark_gizmo* the first null-terminated string is the text of concern. The break gizmo represents a node in the line breaking algorithm, it therefore contains a number of variables relevant in the algorithm. All gizmo widths are measured in thousandths of points.

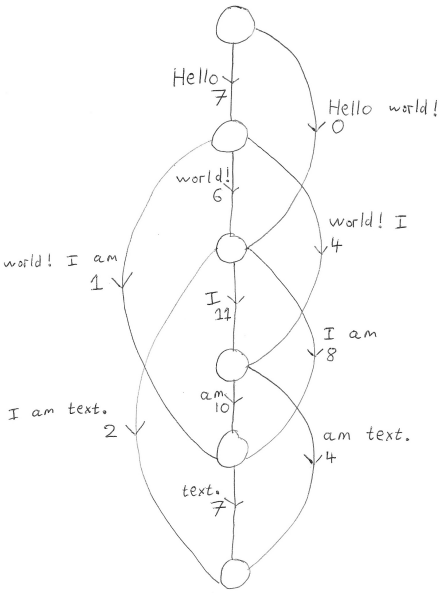
2.6 Key Algorithms

2.6.1 Line Breaker

The line breaking algorithm used in this project is partially based of that presented in Donald E Knuth's 'Breaking Paragraphs into Lines (1981)' [3].

Input text is modelled by a gizmo list defined in section 2.5.4. Consider an directed acyclic graph (DAG) where each vertex is a potential line break and each edge a feasible line of text. The source vertex is the hypothetical line break preceding the body of text and end sink vertex represents the line break after the last item of text. An edge is feasible only if the sum of its text width is less than the maximum line width. Edges are weighted by the difference between the line's text width and the maximum line width. The problem of finding optimal line breaks (that minimises trailing white space) is equivalent to finding the shortest path through this graph from the source to the sink. Therefore, to find optimal line breaks, the algorithm must identify feasible lines, evaluate the weight of these lines and find the shortest path through the graph. These three tasks can be achieved in a single pass.

The shortest path of a DAG can be computed by relaxing each vertex in order of a topological sort [8]. As each edge can only connect an earlier line break to a later one, and because line breaks in the gizmo list appear in text order, the break gizmos are already necessarily in topological order. This means that line breaks can be relaxed in the order they appear in the gizmo list. Relaxing a break involves finding each feasible line that may follow this break, computing the weight of the line and if the destination break does not claim a shorter path, then update the destination breaks shortest path claim to that which ends with this edge.



The diagram **(right)** illustrates the solved graph for the text "Hello world! I am text." with a mono-spaced 1 point font fitting in lines of max width 12. Edges are labeled with their weight and the text contained in the line they represent. As you can see, the shortest route from top to bottom is through, "Hello world!" and "I am text." with a total weight of 2. This means that the optimal line break set is the single break after "world!".

2.6.2 Page Breaker

The purpose of the page breaker is to divide *content* into *pages.* Boxes and glue (both are a type of 'gizmo') are collected sequentially from the input *content* into a normal bin and into a footnote bin. The chosen bin depends on the argument of the most recent flow command. When an optional page break or new page command is reached, if the current page has sufficient space to fit the pending gizmos in both bins, then they are added to the current page. Otherwise, the pending gizmos are added to a new page which becomes the current. After reading a new page command and fitting the pending gizmos, a new page is added. When a mark command is encountered, the page number of the current page is added to the contents file.

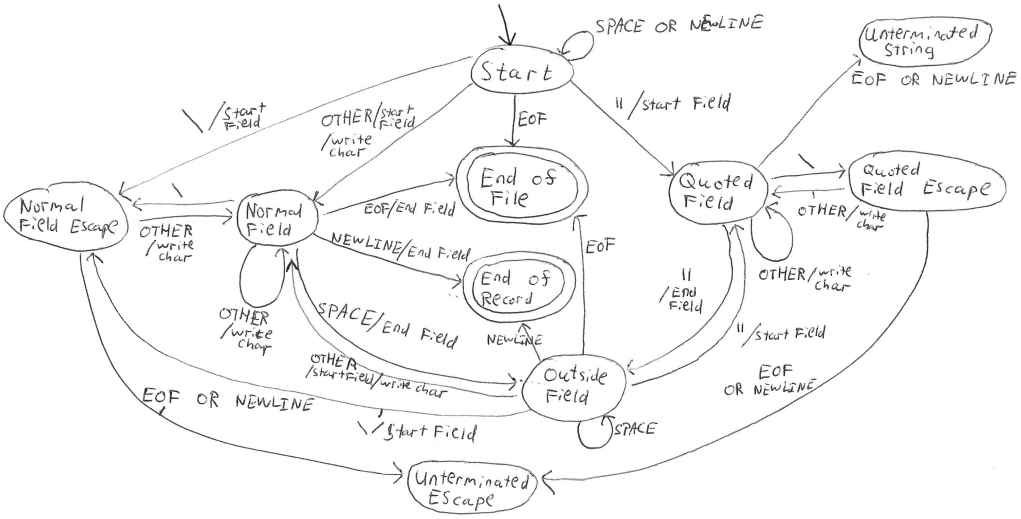
To determine weather a set of gizmos fits on a page, the height of the content of each bin is calculated by summing the height of all gizmos in that bin excluding trailing glue gizmos. If the total height is smaller than the height of the page's max content height (as calculated from the page height and vertical margins) then the gizmos fit.

2.6.3 Record Parser

The python implantation of record parsing uses the following regular expression:

```
[^"\s]\S*|".*?[^\]
```

However, I take more pride in my C code and so in the name of efficiency will be writing a custom parser. Characters are to be parsed sequentially using the following state machine of 10 states.



Parsing begins in the 'Start' state. 'EOF' indicated the End Of File. When a field is started, a pointer to the current end of the *string* buffer is added to the *fields* array. When a character is added to a field, the character is appended to *string*. When a field is ended, a zero byte is appended to *string*. When a terminating or error state is reached, parsing stops immediately. The result of parsing is fields separated and ended in zero bytes in the *string* dynamic buffer and pointers to the beginning of each field in the *fields* array.

2.7 Example Document

This section will consider the following shell command and explain how it should work.

```
echo "Hello world" | markup_text.py -w 60 | pager.py | tw
```

First, "Hello world" is piped into markup_text.py with content width set to 60 points. The string will be parsed and converted into a *text_specification*. markup_text.py will automatically pass this to a new instance of the line_break program. The following is the *text_specification*.

```
FONT Regular 12
STRING "Hello"
OPTBREAK " " "" 0
STRING "world"
```

The line_break program will respond with the following *content* which markup_text.py will output.

```
box 12
START TEXT
FONT Regular 12
STRING "Hello"
END
opt_break
box 12
START TEXT
FONT Regular 12
STRING "world"
END
opt_break
```

pager.py will interpret this *content,* and split it into *pages* and output the following.

```
START PAGE
MOVE 102 705
START TEXT
FONT Regular 12
STRING "Hello"
END
MOVE 102 693
START TEXT
FONT Regular 12
STRING "world"
END
END
```

Finally, tw reads the *pages* and writes the pdf that is described. In this case, a single page with "Hello" on the first line and "world" on the second.

3 Technical Solution

3.1 Techniques Used

Complex File Formats Parsing for the hierarchical *pages* format is implemented in 'tw.c'. I designed this format specifically for this project; it is defined in Documented Design.

Recursive Algorithm 'tw.c' parses the *pages* format by recursivly calling *parse_graphic.*

Dynamic Buffer 'tw.h' defines *struct dbuffer* and 'dbuffer.c' implements functions for it. It allocates a region of memory on the heap. When more memory is required for the buffer, it is reallocated. *dbuffer_printf* can be used to print a formatted string directly to the end of the buffer.

String List 'tw.h' defines *struct record* and 'record.c' implements functions for it. This record stores a list of strings (fields) on the heap and keeps pointers to each of these strings in an array also stored on the heap. *find_field* can be used to search for a string in the record.

Linked List 'line_break.c' defines *struct typeface* and the *gizmo* structures. These are linked lists. Functions to manage these linked lists content and memory are also defined: *open_typeface, free_typeface, parse_gizmos, free_gizmos,* etc.

State Machine 'record.c' uses a state machine to parse records. *enum ParseState* defines the states; *parse_record* implements the state machine and state transitions.

Polymorphism 'line_break.c' defines *struct text_gizmo, struct break_gizmo* and *struct mark_gizmo.* Each of these structures share a number of starting bytes with the same meaning (defined by *struct gizmo*). This means that a function can be passed a generic *struct gizmo* which can be cast to the correct gizmo type based of *type.* Polymorphism is also used in 'pager.py', where *Box* and *Glue* implement the same methods.

Inheritance Inheritance is used in 'markup_text.py' to define a subtypes of TextStream can be defined to change how a stream of text if parsed and converted into content.

Shortest Path in Directed Acyclic Graph This shortest path algorithm is similar to Dijkstra's, but is faster since it takes advantage of the graphs acyclic nature and topological sort. It finished in O(e+v) time where e is number of edges and v is number of vertices. It is implemented in 'line_break.c' to determine the optimal line breaks that minimise the total trailing whitespace at the end of each line.

Exception Handling Thorough exception handling is used throughout the codebase. Examples can be found in most source files. One specific example is at the end of *parse_record* in 'record.c' where unterminated strings and escapes are handled.

File Paths Parameterised 'tw.c' provides a command line argument to change the output PDF filename. 'pager.py' provides a command line argument to change the

output 'contents' filename.

Symbolic Links Symbolic links are used in the 'test' and 'report' folders to resolve the typeface file and fonts directory.

3.2 Source Code

Makefile

```
001 CC=gcc
002 CFLAGS=-g -Wall
003
004 all: tw line_break
005
006 tw: tw.o utils.o dbuffer.o record.o ttf.o jpeg.o pdf.o
007     $(CC) $^ -o $@
008
009 line_break: line_break.o utils.o dbuffer.o record.o ttf.o
010     $(CC) $^ -o $@
011
012 tw.o: tw.c tw.h
013     $(CC) $(CFLAGS) -c $< -o $@
014
015 line_break.o: line_break.c tw.h
016     $(CC) $(CFLAGS) -c $< -o $@
017
018 utils.o: utils.c tw.h
019     $(CC) $(CFLAGS) -c $< -o $@
020
021 dbuffer.o: dbuffer.c tw.h
022     $(CC) $(CFLAGS) -c $< -o $@
023
024 record.o: record.c tw.h
025     $(CC) $(CFLAGS) -c $< -o $@
026
027 ttf.o: ttf.c tw.h
028     $(CC) $(CFLAGS) -c $< -o $@
029
030 jpeg.o: jpeg.c tw.h
031     $(CC) $(CFLAGS) -c $< -o $@
032
033 pdf.o: pdf.c tw.h
034     $(CC) $(CFLAGS) -c $< -o $@
```

tw.h

```
001 /*
002  * Copyright (C) 2023 Christopher Lang
003  * See LICENSE for license details.
004  */
005
006 /* Dynamic Buffer */
007 struct dbuffer {
008   int size, allocated, increment;
009   char *data;
010 };
011
012 struct record {
013   /* _string_ is for zero byte seperated and ended field strings. */
014   struct dbuffer string;
015   int field_count;
016   int fields_allocated;
017   /* _fields_ is an array of pointers to each field string in _string_. */
018   const char **fields;
019 };
020
021 struct font_info {
022   int units_per_em;
023   int x_min;
024   int y_min;
025   int x_max;
026   int y_max;
```

```
027   int long_hor_metrics_count;
028   int cmap[256];
029   int char_widths[256];
030 };
031
032 struct jpeg_info {
033   int width, height;
034   unsigned char components;
035 };
036
037 struct pdf_resources {
038   /* Each field in _fonts_used_ is a font name that is used in the PDF. */
039   struct record fonts_used;
040   /* Each field in _image_ is a image file name that is used in the PDF. */
041   struct record images;
042 };
043
044 struct pdf_page_list {
045   int page_count, pages_allocated;
046   int *page_objs;
047 };
048
049 struct pdf_xref_table {
050   int obj_count, allocated;
051   long *obj_offsets;
052 };
053
054 /* utils.c */
055 void *xmalloc(size_t len);
056 void *xrealloc(void *p, size_t len);
057 int is_font_name_valid(const char *font_name);
058 int str_to_int(const char *str, int *n);
059
060 /* dbuffer.c */
061 void dbuffer_init(struct dbuffer *buf, int initial, int increment);
062 void dbuffer_putc(struct dbuffer *buf, char c);
063 void dbuffer_printf(struct dbuffer *buf, const char *format, ...);
064 void dbuffer_free(struct dbuffer *buf);
065
066 /* record.c */
067 void init_record(struct record *record);
068 void begin_field(struct record *record);
069 int parse_record(FILE *file, struct record *record);
070 int find_field(const struct record *record, const char *field_str);
071 void free_record(struct record *record);
072
073 /* ttf.c */
074 int read_ttf(FILE *file, struct font_info *info);
075
076 /* jpeg.c */
077 int read_jpeg(FILE *file, struct jpeg_info *info);
078
079 /* pdf.c */
080 void pdf_write_header(FILE *file);
081 void pdf_start_indirect_obj(FILE *file, struct pdf_xref_table *xref, int obj);
082 void pdf_end_indirect_obj(FILE *file);
083 void pdf_write_file_stream(FILE *pdf_file, FILE *data_file);
084 void pdf_write_text_stream(FILE *file, const char *data, long size);
085 void pdf_write_int_array(FILE *file, const int *values, int count);
086 void pdf_write_font_descriptor(FILE *file, int font_file, const char *font_name,
087     int italic_angle, int ascent, int descent, int cap_height,
088     int stem_vertical, int min_x, int min_y, int max_x, int max_y);
089 void pdf_write_page(FILE *file, int parent, int content);
090 void pdf_write_font(FILE *file, const char *font_name, int font_descriptor,
091     int font_widths);
092 void pdf_write_pages(FILE *file, int resources, int page_count,
093     const int *page_objs);
094 void pdf_write_catalog(FILE *file, int page_list);
095 void init_pdf_xref_table(struct pdf_xref_table *xref);
096 int allocate_pdf_obj(struct pdf_xref_table *xref);
097 void pdf_add_footer(FILE *file, const struct pdf_xref_table *xref,
098     int root_obj);
099 void free_pdf_xref_table(struct pdf_xref_table *xref);
100 void init_pdf_resources(struct pdf_resources *resources);
```

```
101 void include_font_resource(struct pdf_resources *resources, const char *font);
102 int include_image_resource(struct pdf_resources *resources, const char *fname);
103 void pdf_add_resources(FILE *pdf_file, FILE *typeface_file, int resources_obj,
104     const struct pdf_resources *resources, struct pdf_xref_table *xref);
105 void free_pdf_resources(struct pdf_resources *resources);
106 void init_pdf_page_list(struct pdf_page_list *page_list);
107 void add_pdf_page(struct pdf_page_list *page_list, int page);
108 void free_pdf_page_list(struct pdf_page_list *page_list);
109
110 /* print_pages.c */
111 int print_pages(FILE *pages_file, FILE *font_file, FILE *pdf_file);
```

tw.c

```
001 /*
002  * Copyright (C) 2023 Christopher Lang
003  * See LICENSE for license details.
004  */
005
006 #include <stdio.h>
007 #include <string.h>
008 #include <unistd.h>
009
010 #include "tw.h"
011
012 /* A page's entire text content and its current state. */
013 struct text_content {
014   int x, y, font_size, word_spacing;
015   struct dbuffer buffer;
016   char font_name[256];
017 };
018
019 static void add_page(FILE *pdf_file, int obj_parent,
020     struct pdf_xref_table *xref, struct pdf_page_list *page_list,
021     struct dbuffer *text_content, struct dbuffer *graphic_content);
022 static void write_pdf_escaped_string(struct dbuffer *buffer,
023     const char *string);
024 static int parse_text(FILE *input, int x, int y,
025     struct text_content *text_content, struct pdf_resources *resources);
026 static int parse_graphic(FILE *input, int origin_x, int origin_y,
027     struct text_content *text_content, struct dbuffer *graphic_content,
028     struct pdf_resources *resources);
029
030 /*
031  * Records are parsed in multiple functions, in order to avoid reallocating
032  * record memory each time, _record_ is kept as a static global variable.
033  * It is reset before each use so its persistent state does not matter.
034  * Use of record is not thread-safe.
035  */
036 static struct record record;
037
038 /* Add a page to _pdf_file_ with _text_content_ and _graphic_content_. */
039 static void
040 add_page(FILE *pdf_file, int obj_parent, struct pdf_xref_table *xref,
041     struct pdf_page_list *page_list, struct dbuffer *text_content,
042     struct dbuffer *graphic_content)
043 {
044   int obj_content, obj_page;
045   long length;
046   obj_content = allocate_pdf_obj(xref);
047   obj_page = allocate_pdf_obj(xref);
048
049   /* ET is the PDF control sequence for ending text content. */
050   dbuffer_printf(text_content, "ET\n");
051   /* Write a PDF stream object containing the page content. */
052   length = text_content->size + graphic_content->size;
053   pdf_start_indirect_obj(pdf_file, xref, obj_content);
054   fprintf(pdf_file, "<< /Length %ld >> stream\n", length);
055   fwrite(text_content->data, 1, text_content->size, pdf_file);
056   fwrite(graphic_content->data, 1, graphic_content->size, pdf_file);
057   fprintf(pdf_file, "\nendstream\n");
058   pdf_end_indirect_obj(pdf_file);
059
```

```
060   /*
061    * The PDF page object holds a reference to the PDF page content stream
062    * object.
063    */
064   pdf_start_indirect_obj(pdf_file, xref, obj_page);
065   pdf_write_page(pdf_file, obj_parent, obj_content);
066   pdf_end_indirect_obj(pdf_file);
067
068   /* Save a reference to the PDF page object. */
069   add_pdf_page(page_list, obj_page);
070 }
071
072 /* Write a PDF specification compliant string to _buffer_. */
073 static void
074 write_pdf_escaped_string(struct dbuffer *buffer, const char *string)
075 {
076   dbuffer_putc(buffer, '(');
077   while (*string) {
078     switch (*string) {
079       case '(':
080       case ')':
081       case '\\':
082         dbuffer_putc(buffer, '\\');
083       default:
084         dbuffer_putc(buffer, *string);
085     }
086     string++;
087   }
088   dbuffer_putc(buffer, ')');
089 }
090
091 /* Parse text mode _pages_ content. */
092 static int
093 parse_text(FILE *input, int x, int y, struct text_content *text_content,
094     struct pdf_resources *resources)
095 {
096   int parse_result, arg1, font_size, word_spacing;
097   char font_name[256];
098   font_size = 0;
099   font_name[0] = '\0';
100   word_spacing = 0;
101   /* Set the text transformation matrix to the (x,y) transformation. */
102   dbuffer_printf(&text_content->buffer, "1 0 0 1 %d %d Tm", x, y);
103   for (;;) {
104     parse_result = parse_record(input, &record);
105     if (parse_result == EOF) {
106       fprintf(stderr, "Text not ended before end of file.\n");
107       return 1;
108     }
109     if (parse_result) /* If failed to parse record then skip it. */
110       continue;
111     if (strcmp(record.fields[0], "END") == 0)
112       /* Exit text mode. */
113       break;
114     if (strcmp(record.fields[0], "FONT") == 0) {
115       /* FONT [FONT_NAME] [FONT_SIZE] */
116       if (record.field_count != 3) {
117         fprintf(stderr, "Text FONT command must take 2 arguments.\n");
118         continue;
119       }
120       if (strlen(record.fields[1]) >= 256) {
121         fprintf(stderr, "Font name too long: '%s'\n", record.fields[1]);
122         continue;
123       }
124       if (!is_font_name_valid(record.fields[1])) {
125         fprintf(stderr, "Font name contains illegal characters: '%s'\n",
126             record.fields[1]);
127         continue;
128       }
129       if (str_to_int(record.fields[2], &arg1)) {
130         fprintf(stderr, "Text FONT command's 2nd argument must be integer.\n");
131         continue;
132       }
133       strcpy(font_name, record.fields[1]);
```

```
134       font_size = arg1;
135       include_font_resource(resources, font_name);
136       continue;
137     }
138     if (strcmp(record.fields[0], "SPACE") == 0) {
139       /* SPACE [WORD_SPACING] */
140       if (record.field_count != 2) {
141         fprintf(stderr, "Text SPACE command must take 1 arguments.\n");
142         continue;
143       }
144       if (str_to_int(record.fields[1], &arg1)) {
145         fprintf(stderr, "Text SPACE command's argument must be integer.\n");
146         continue;
147       }
148       word_spacing = arg1;
149       continue;
150     }
151     if (strcmp(record.fields[0], "STRING") == 0) {
152       /* STRING [STRING] */
153       if (record.field_count != 2) {
154         fprintf(stderr, "Text STRING command must take 1 argument.\n");
155         continue;
156       }
157       if (*font_name == '\0') {
158         fprintf(stderr, "Text must specify font.\n");
159         continue;
160       }
161       if (strcmp(text_content->font_name, font_name)
162           || text_content->font_size != font_size) {
163         dbuffer_printf(&text_content->buffer, " /%s %d Tf", font_name,
164             font_size);
165         strcpy(text_content->font_name, font_name);
166         text_content->font_size = font_size;
167       }
168       if (word_spacing != text_content->word_spacing) {
169         dbuffer_printf(&text_content->buffer, " %f Tw",
170             (float)word_spacing / 1000);
171         text_content->word_spacing = word_spacing;
172       }
173       dbuffer_putc(&text_content->buffer, ' ');
174       write_pdf_escaped_string(&text_content->buffer, record.fields[1]);
175       dbuffer_printf(&text_content->buffer, " Tj");
176       continue;
177     }
178     fprintf(stderr, "Invalid text command: '%s'\n", record.fields[0]);
179   }
180   dbuffer_putc(&text_content->buffer, '\n');
181   return 0;
182 }
183
184 /* Parse graphic mode _pages_ content. */
185 static int
186 parse_graphic(FILE *input, int origin_x, int origin_y,
187     struct text_content *text_content, struct dbuffer *graphic_content,
188     struct pdf_resources *resources)
189 {
190   int parse_result, x, y, arg1, arg2, image_id;
191   x = origin_x;
192   y = origin_y;
193   for (;;) {
194     parse_result = parse_record(input, &record);
195     if (parse_result == EOF) {
196       fprintf(stderr, "Graphic not ended before end of file.\n");
197       return 1;
198     }
199     if (parse_result) /* If failed to parse record then skip it. */
200       continue;
201     if (strcmp(record.fields[0], "END") == 0)
202       /* Exit graphic mode. */
203       break;
204     if (strcmp(record.fields[0], "MOVE") == 0) {
205       /* MOVE [X_OFFSET] [Y_OFFSET] */
206       if (record.field_count != 3) {
207         fprintf(stderr, "Graphic MOVE command must take 2 arguments.\n");
```

```
208         continue;
209       }
210       if (str_to_int(record.fields[1], &arg1)
211           || str_to_int(record.fields[2], &arg2)) {
212         fprintf(stderr, "Graphic MOVE command takes only integer arguments.\n");
213         continue;
214       }
215       x = origin_x + arg1;
216       y = origin_y + arg2;
217       continue;
218     }
219     if (strcmp(record.fields[0], "START") == 0) {
220       /* START [GRAPHIC/TEXT] */
221       /* A graphic may contain text content or another graphic. */
222       if (record.field_count != 2) {
223         fprintf(stderr, "START command must take one argument.\n");
224         return 1;
225       }
226       if (strcmp(record.fields[1], "GRAPHIC") == 0) {
227         if (parse_graphic(input, x, y, text_content, graphic_content,
228               resources))
229           return 1;
230         continue;
231       }
232       if (strcmp(record.fields[1], "TEXT") == 0) {
233         if (parse_text(input, x, y, text_content, resources))
234           return 1;
235         continue;
236       }
237       fprintf(stderr, "Invalid graphic START command argument: '%s'\n",
238           record.fields[1]);
239       return 1;
240     }
241     if (strcmp(record.fields[0], "IMAGE") == 0) {
242       /* IMAGE [WIDTH] [HEIGHT] [IMAGE_FILE_NAME] */
243       if (record.field_count != 4) {
244         fprintf(stderr, "IMAGE command must take 3 arguments.\n");
245         continue;
246       }
247       if (str_to_int(record.fields[1], &arg1)
248           || str_to_int(record.fields[2], &arg2)) {
249         fprintf(stderr,
250             "IMAGE command's 1st and 2nd arguments must be integer.\n");
251         continue;
252       }
253       image_id = include_image_resource(resources, record.fields[3]);
254       /* Push graphic state. */
255       dbuffer_printf(graphic_content, "q\n");
256       /* Set graphic transformation matrix and draw image. */
257       dbuffer_printf(graphic_content, "  %d 0 0 %d %d %d cm\n", arg1, arg2, x,
258           y);
259       dbuffer_printf(graphic_content, "  /Img%d Do\n", image_id);
260       /* Pop graphic state. */
261       dbuffer_printf(graphic_content, "Q\n");
262       continue;
263     }
264     fprintf(stderr, "Invalid graphic command: '%s'\n", record.fields[0]);
265   }
266   return 0;
267 }
268
269 /* Read _pages_ format from _pages_file_ and write PDF to _pdf_file_. */
270 int
271 print_pages(FILE *pages_file, FILE *typeface_file, FILE *pdf_file)
272 {
273   int ret, parse_result;
274   int obj_resources, obj_page_list, obj_catalog;
275   struct pdf_xref_table xref_table;
276   struct pdf_page_list page_list;
277   struct pdf_resources resources;
278   struct text_content text_content;
279   struct dbuffer graphic_content;
280   ret = 0;
281   init_pdf_xref_table(&xref_table);
```

```
282   init_pdf_page_list(&page_list);
283   init_pdf_resources(&resources);
284   obj_resources = allocate_pdf_obj(&xref_table);
285   obj_page_list = allocate_pdf_obj(&xref_table);
286   obj_catalog = allocate_pdf_obj(&xref_table);
287   pdf_write_header(pdf_file);
288
289   text_content.x = 0;
290   text_content.y = 0;
291   text_content.font_size = 0;
292   text_content.font_name[0] = '\0';
293   dbuffer_init(&text_content.buffer, 1024 * 32, 1024 * 32);
294   dbuffer_printf(&text_content.buffer, "BT\n");
295   dbuffer_init(&graphic_content, 1024 * 4, 1024 * 4);
296   init_record(&record);
297   /* Parse document mode _pages_. */
298   for (;;) {
299     parse_result = parse_record(pages_file, &record);
300     if (parse_result == EOF)
301       break;
302     if (parse_result)
303       continue;
304     if (strcmp(record.fields[0], "START") == 0) {
305       if (record.field_count != 2) {
306         fprintf(stderr, "Pages START command must take 1 argument.\n");
307         ret = 1;
308         break;
309       }
310       if (strcmp(record.fields[1], "PAGE") == 0) {
311         if (parse_graphic(pages_file, 0, 0, &text_content, &graphic_content,
312               &resources)) {
313           ret = 1;
314           break;
315         }
316         add_page(pdf_file, obj_page_list, &xref_table, &page_list,
317             &text_content.buffer, &graphic_content);
318         text_content.x = 0;
319         text_content.y = 0;
320         text_content.font_size = 0;
321         text_content.font_name[0] = '\0';
322         text_content.buffer.size = 0;
323         dbuffer_printf(&text_content.buffer, "BT\n");
324         graphic_content.size = 0;
325         graphic_content.data[0] = '\0';
326         continue;
327       }
328       fprintf(stderr, "Invalid document START command argument: '%s'\n",
329           record.fields[1]);
330       ret = 1;
331       break;
332     }
333   }
334   dbuffer_free(&text_content.buffer);
335   free_record(&record);
336
337   pdf_add_resources(pdf_file, typeface_file, obj_resources, &resources,
338       &xref_table);
339
340   pdf_start_indirect_obj(pdf_file, &xref_table, obj_page_list);
341   pdf_write_pages(pdf_file, obj_resources, page_list.page_count,
342       page_list.page_objs);
343   pdf_end_indirect_obj(pdf_file);
344
345   pdf_start_indirect_obj(pdf_file, &xref_table, obj_catalog);
346   pdf_write_catalog(pdf_file, obj_page_list);
347   pdf_end_indirect_obj(pdf_file);
348
349   pdf_add_footer(pdf_file, &xref_table, obj_catalog);
350   free_pdf_page_list(&page_list);
351   free_pdf_resources(&resources);
352   free_pdf_xref_table(&xref_table);
353   return ret;
354 }
355
```

```
356 int
357 main(int argc, char **argv)
358 {
359   int opt;
360   const char *output_file_name;
361   FILE *pages_file, *typeface_file, *pdf_file;
362   output_file_name = "./output.pdf";
363   while ( (opt = getopt(argc, argv, "o:")) != -1) {
364     switch (opt) {
365       case 'o':
366         output_file_name = optarg;
367         break;
368       default:
369         fprintf(stderr, "Usage: %s [-o OUTPUT_FILE]\n", argv[0]);
370         return 1;
371     }
372   }
373   pages_file = stdin;
374   typeface_file = fopen("./typeface", "r");
375   pdf_file = fopen(output_file_name, "w");
376   if (typeface_file == NULL) {
377     fprintf(stderr, "tw: Failed to open typeface file.\n");
378     return 1;
379   }
380   if (pdf_file == NULL) {
381     fprintf(stderr, "tw: Failed to open output file '%s'.\n", output_file_name);
382     return 1;
383   }
384   print_pages(pages_file, typeface_file, pdf_file);
385   fclose(pages_file);
386   fclose(typeface_file);
387   fclose(pdf_file);
388   return 0;
389 }
```

line_break.c

```
001 /*
002  * Copyright (C) 2023 Christopher Lang
003  * See LICENSE for license details.
004  */
005
006 #include <stdio.h>
007 #include <string.h>
008 #include <stdlib.h>
009 #include <errno.h>
010 #include <limits.h>
011 #include <unistd.h>
012
013 #include "tw.h"
014
015 enum align {
016   ALIGN_LEFT,
017   ALIGN_RIGHT,
018   ALIGN_CENTRE,
019   ALIGN_JUSTIFIED,
020 };
021
022 enum gizmo_type {
023   GIZMO_TEXT,
024   GIZMO_BREAK,
025   GIZMO_MARK,
026 };
027
028 /* Linked list, each struct maps one _font_name_ to one _font_info_. */
029 struct typeface {
030   char font_name[256];
031   struct font_info font_info;
032   struct typeface *next;
033 };
034
035 struct style {
036   int font_size;
```

```
037   char font_name[256];
038 };
039
040 /* Polymorphic linked list. _type_ defines the meaning of _char _[]_ bytes. */
041 struct gizmo {
042   int type;
043   struct gizmo *next;
044   char _[];
045 };
046
047 struct text_gizmo {
048   int type; /* GIZMO_TEXT */
049   struct gizmo *next;
050   int width;
051   struct style style;
052   char string[];
053 };
054
055 struct break_gizmo {
056   int type; /* GIZMO_BREAK */
057   struct gizmo *next;
058   int force_break, total_penalty, spacing, selected;
059   struct break_gizmo *best_source;
060   struct style style;
061   int no_break_width, at_break_width;
062   char *no_break, *at_break; /* Pointers to inside _strings_. */
063   char strings[];
064 };
065
066 struct mark_gizmo {
067   int type; /* GIZMO_MARK */
068   struct gizmo *next;
069   char string[];
070 };
071
072 static struct typeface *open_typeface(FILE *typeface_file);
073 static void free_typeface(struct typeface *typeface);
074 static int get_text_width(const char *string, const struct style *style,
075     const struct typeface *typeface);
076 static struct gizmo *parse_gizmos(FILE *file, const struct typeface *typeface);
077 static void free_gizmos(struct gizmo *gizmo);
078 static void consider_breaks(struct gizmo *gizmo, int source_penalty,
079     struct break_gizmo *source, int line_width);
080 static void optimise_breaks(struct gizmo *gizmo, int line_width);
081 static void print_text(struct dbuffer *buffer, struct style *style,
082     const struct style *new_style, const char *string, int *spaces);
083 static void print_gizmos(FILE *output, struct gizmo *gizmo, int line_width,
084     int align);
085
086 /*
087  * Parse the _typeface_file_ and return a linked list of fonts in the typeface.
088  */
089 static struct typeface *
090 open_typeface(FILE *typeface_file)
091 {
092   struct typeface *first_font;
093   struct typeface **next_font;
094   int parse_result;
095   struct record record;
096   FILE *font_file;
097   first_font = NULL;
098   next_font = &first_font;
099   init_record(&record);
100   for (;;) {
101     parse_result = parse_record(typeface_file, &record);
102     if (parse_result == EOF) /* Reached end of file. */
103       break;
104     if (parse_result) /* If failed to parse this record then skip it. */
105       continue;
106     if (record.field_count != 2) {
107       fprintf(stderr, "Typeface records must have exactly 2 fields.");
108       continue;
109     }
110     if (strlen(record.fields[0]) >= 256) {
```

```
111       fprintf(stderr,
112           "Typeface file contains font name that is too long '%s'.\n",
113           record.fields[0]);
114       continue;
115     }
116     if (!is_font_name_valid(record.fields[0])) {
117       fprintf(stderr, "Typeface file contains invalid font name '%s'.\n",
118           record.fields[0]);
119       continue;
120     }
121     font_file = fopen(record.fields[1], "r");
122     if (font_file == NULL) {
123       fprintf(stderr, "Failed to open ttf file '%s': %s\n",
124           record.fields[1], strerror(errno));
125       continue;
126     }
127     *next_font = xmalloc(sizeof(struct typeface));
128     strcpy((*next_font)->font_name, record.fields[0]);
129     if (read_ttf(font_file, &(*next_font)->font_info)) {
130       fprintf(stderr, "Failed to parse ttf file: '%s'\n", record.fields[1]);
131       free(*next_font);
132     } else {
133       next_font = &(*next_font)->next;
134     }
135     fclose(font_file);
136   }
137   *next_font = NULL;
138   free_record(&record);
139   return first_font;
140 }
141
142 static void
143 free_typeface(struct typeface *typeface)
144 {
145   struct typeface *next;
146   while (typeface) {
147     next = typeface->next;
148     free(typeface);
149     typeface = next;
150   }
151 }
152
153 /*
154  * Compute the width of _string_ in _style_ with _typeface_ measured in
155  * thousandths of points.
156  */
157 static int
158 get_text_width(const char *string, const struct style *style,
159     const struct typeface *typeface)
160 {
161   int width;
162   const struct typeface *font;
163   font = typeface;
164   while (font && strcmp(font->font_name, style->font_name))
165     font = font->next;
166   if (font == NULL) {
167     fprintf(stderr, "Typeface does not include font: '%s'\n", style->font_name);
168     font = typeface;
169   }
170   width = 0;
171   for (; *string; string++)
172     width += font->font_info.char_widths[(unsigned char)*string];
173   return width * style->font_size;
174 }
175
176 /*
177  * Parse text gizmos from the text specification _file_ and return them as a
178  * linked list.
179  */
180 static struct gizmo *
181 parse_gizmos(FILE *file, const struct typeface *typeface)
182 {
183   struct gizmo *first_gizmo;
184   struct gizmo **next_gizmo;
```

```
185   struct style current_style;
186   int parse_result, arg1;
187   struct record record;
188   first_gizmo = NULL;
189   next_gizmo = &first_gizmo;
190   current_style.font_name[0] = '\0';
191   current_style.font_size = 0;
192   init_record(&record);
193   for (;;) {
194     parse_result = parse_record(file, &record);
195     if (parse_result == EOF)
196       break;
197     if (parse_result) /* If failed to parse record then skip it. */
198       continue;
199     if (strcmp(record.fields[0], "STRING") == 0) {
200       /* STRING [STRING] */
201       if (record.field_count != 2) {
202         fprintf(stderr, "Text STRING command must have 1 option.\n");
203         continue;
204       }
205       if (current_style.font_name[0] == '\0') {
206         fprintf(stderr,
207             "Text STRING command can't be called without FONT set.\n");
208         continue;
209       }
210       *next_gizmo = xmalloc(sizeof(struct text_gizmo)
211           + strlen(record.fields[1]) + 1);
212       (*next_gizmo)->type = GIZMO_TEXT;
213       (*next_gizmo)->next = NULL;
214       ((struct text_gizmo *)*next_gizmo)->width
215         = get_text_width(record.fields[1], &current_style, typeface);
216       ((struct text_gizmo *)*next_gizmo)->style = current_style;
217       strcpy(((struct text_gizmo *)*next_gizmo)->string, record.fields[1]);
218       next_gizmo = &(*next_gizmo)->next;
219       continue;
220     }
221     if (strcmp(record.fields[0], "FONT") == 0) {
222       /* FONT [FONT_NAME] [FONT_SIZE] */
223       if (record.field_count != 3) {
224         fprintf(stderr, "Text FONT command must have 2 options.\n");
225         continue;
226       }
227       if (strlen(record.fields[1]) >= 256) {
228         fprintf(stderr, "Text font name too long: '%s'\n", record.fields[1]);
229         continue;
230       }
231       if (!is_font_name_valid(record.fields[1])) {
232         fprintf(stderr, "Text font name contains illegal characters: '%s'\n",
233             record.fields[1]);
234         continue;
235       }
236       strcpy(current_style.font_name, record.fields[1]);
237       if (str_to_int(record.fields[2], &current_style.font_size)) {
238         fprintf(stderr, "Text FONT command's 2nd option must be integer.\n");
239         current_style.font_size = 12;
240       }
241       continue;
242     }
243     if (strcmp(record.fields[0], "OPTBREAK") == 0) {
244       /* OPTBREAK [NO_BREAK_STRING] [AT_BREAK_STRING] [SPACING] */
245       if (record.field_count != 4) {
246         fprintf(stderr, "Text OPTBREAK command must have 3 options.\n");
247         continue;
248       }
249       if (str_to_int(record.fields[3], &arg1)) {
250         fprintf(stderr,
251             "Text OPTBREAK command's 3rd option must be integer.\n");
252         continue;
253       }
254       *next_gizmo = xmalloc(sizeof(struct break_gizmo)
255           + strlen(record.fields[1])
256           + strlen(record.fields[2]) + 2);
257       (*next_gizmo)->type = GIZMO_BREAK;
258       (*next_gizmo)->next = NULL;
```

```
259       ((struct break_gizmo *)*next_gizmo)->spacing = arg1;
260       ((struct break_gizmo *)*next_gizmo)->force_break = 0;
261       ((struct break_gizmo *)*next_gizmo)->selected = 0;
262       ((struct break_gizmo *)*next_gizmo)->total_penalty = INT_MAX;
263       ((struct break_gizmo *)*next_gizmo)->best_source = NULL;
264       ((struct break_gizmo *)*next_gizmo)->style = current_style;
265       ((struct break_gizmo *)*next_gizmo)->no_break_width
266         = get_text_width(record.fields[1], &current_style, typeface);
267       ((struct break_gizmo *)*next_gizmo)->at_break_width
268         = get_text_width(record.fields[2], &current_style, typeface);
269       ((struct break_gizmo *)*next_gizmo)->no_break
270         = ((struct break_gizmo *)*next_gizmo)->strings;
271       ((struct break_gizmo *)*next_gizmo)->at_break
272         = ((struct break_gizmo *)*next_gizmo)->strings
273         + strlen(record.fields[1]) + 1;
274       strcpy(((struct break_gizmo *)*next_gizmo)->no_break,
275           record.fields[1]);
276       strcpy(((struct break_gizmo *)*next_gizmo)->at_break,
277           record.fields[2]);
278       next_gizmo = &(*next_gizmo)->next;
279       continue;
280     }
281     if (strcmp(record.fields[0], "BREAK") == 0) {
282       /* BREAK [SPACING] */
283       if (record.field_count != 2) {
284         fprintf(stderr, "Text BREAK command must take 1 option.\n");
285         continue;
286       }
287       if (str_to_int(record.fields[1], &arg1)) {
288         fprintf(stderr, "Text BREAK command's 1st option must be integer.\n");
289         continue;
290       }
291       *next_gizmo = xmalloc(sizeof(struct break_gizmo) + 1);
292       (*next_gizmo)->type = GIZMO_BREAK;
293       (*next_gizmo)->next = NULL;
294       ((struct break_gizmo *)*next_gizmo)->spacing = arg1;
295       ((struct break_gizmo *)*next_gizmo)->force_break = 1;
296       ((struct break_gizmo *)*next_gizmo)->selected = 0;
297       ((struct break_gizmo *)*next_gizmo)->total_penalty = INT_MAX;
298       ((struct break_gizmo *)*next_gizmo)->best_source = NULL;
299       ((struct break_gizmo *)*next_gizmo)->style = current_style;
300       ((struct break_gizmo *)*next_gizmo)->no_break_width = 0;
301       ((struct break_gizmo *)*next_gizmo)->at_break_width = 0;
302       ((struct break_gizmo *)*next_gizmo)->no_break
303         = ((struct break_gizmo *)*next_gizmo)->at_break
304         = ((struct break_gizmo *)*next_gizmo)->strings;
305       ((struct break_gizmo *)*next_gizmo)->strings[0] = '\0';
306       next_gizmo = &(*next_gizmo)->next;
307       continue;
308     }
309     if (strcmp(record.fields[0], "MARK") == 0) {
310       /* MARK [MARK_STRING] */
311       if (record.field_count != 2) {
312         fprintf(stderr, "Text MARK command must have 1 option.\n");
313         continue;
314       }
315       *next_gizmo = xmalloc(sizeof(struct mark_gizmo)
316           + strlen(record.fields[1]) + 1);
317       (*next_gizmo)->type = GIZMO_MARK;
318       (*next_gizmo)->next = NULL;
319       strcpy(((struct mark_gizmo *)*next_gizmo)->string, record.fields[1]);
320       next_gizmo = &(*next_gizmo)->next;
321       continue;
322     }
323   }
324   /* A body of text must end in a forced break (the sink of the graph). */
325   *next_gizmo = xmalloc(sizeof(struct break_gizmo) + 1);
326   (*next_gizmo)->type = GIZMO_BREAK;
327   (*next_gizmo)->next = NULL;
328   ((struct break_gizmo *)*next_gizmo)->spacing = 0;
329   ((struct break_gizmo *)*next_gizmo)->force_break = 1;
330   ((struct break_gizmo *)*next_gizmo)->selected = 0;
331   ((struct break_gizmo *)*next_gizmo)->total_penalty = INT_MAX;
332   ((struct break_gizmo *)*next_gizmo)->best_source = NULL;
```

```
333   ((struct break_gizmo *)*next_gizmo)->style = current_style;
334   ((struct break_gizmo *)*next_gizmo)->no_break_width = 0;
335   ((struct break_gizmo *)*next_gizmo)->at_break_width = 0;
336   ((struct break_gizmo *)*next_gizmo)->no_break
337     = ((struct break_gizmo *)*next_gizmo)->at_break
338     = ((struct break_gizmo *)*next_gizmo)->strings;
339   ((struct break_gizmo *)*next_gizmo)->strings[0] = '\0';
340   free_record(&record);
341   return first_gizmo;
342 }
343
344 static void
345 free_gizmos(struct gizmo *gizmo)
346 {
347   struct gizmo *next;
348   while (gizmo) {
349     next = gizmo->next;
350     free(gizmo);
351     gizmo = next;
352   }
353 }
354
355 /* Relax the edges of the _source_ node. */
356 static void
357 consider_breaks(struct gizmo *gizmo, int source_penalty,
358     struct break_gizmo *source, int line_width)
359 {
360   struct break_gizmo *break_gizmo;
361   int width, penalty;
362   width = 0;
363   for (; gizmo; gizmo = gizmo->next) {
364     switch (gizmo->type) {
365     case GIZMO_TEXT:
366       width += ((struct text_gizmo *)gizmo)->width;
367       break;
368     case GIZMO_BREAK:
369       break_gizmo = (struct break_gizmo *)gizmo;
370       /* If feasible line. */
371       if (width + break_gizmo->at_break_width <= line_width) {
372         penalty = (line_width - width - break_gizmo->at_break_width) / 1000;
373         if (break_gizmo->force_break)
374           penalty = 0;
375         if (break_gizmo->total_penalty > source_penalty + penalty) {
376           break_gizmo->best_source = source;
377           break_gizmo->total_penalty = source_penalty + penalty;
378         }
379       }
380       width += break_gizmo->no_break_width;
381       /* If all subsequent lines will not be feasible then stop. */
382       if (width > line_width)
383         goto stop;
384       if (width && break_gizmo->force_break)
385         goto stop;
386       break;
387     }
388   }
389 stop:
390 }
391
392 /* Find the optimal breaks in the linked list starting with _gizmo_. */
393 static void
394 optimise_breaks(struct gizmo *gizmo, int line_width)
395 {
396   struct break_gizmo *last_break;
397   last_break = NULL;
398   /* Relax the hypothetical leading break node. */
399   consider_breaks(gizmo, 0, NULL, line_width);
400   /* Relax every subsequent break node in topological order. */
401   for (; gizmo; gizmo = gizmo->next)
402     if (gizmo->type == GIZMO_BREAK) {
403       last_break = (struct break_gizmo *)gizmo;
404       consider_breaks(gizmo->next, last_break->total_penalty, last_break,
405           line_width);
406     }
```

```
407   /* Backtrack through the graph to set _selected_ to 1 on all chosen breaks. */
408   for (; last_break; last_break = last_break->best_source)
409     last_break->selected = 1;
410 }
411
412 /*
413  * Add new _string_ with _new_style_ to _buffer_'s _pages_ text mode content.
414  * _style_ is updated to match the font state of the text mode content.
415  */
416 static void
417 print_text(struct dbuffer *buffer, struct style *style,
418     const struct style *new_style, const char *string, int *spaces)
419 {
420   if (*string == '\0')
421     return;
422   if (strcmp(style->font_name, new_style->font_name)
423       || style->font_size != new_style->font_size) {
424     memcpy(style, new_style, sizeof(struct style));
425     dbuffer_printf(buffer, "FONT %s %d\n", style->font_name, style->font_size);
426   }
427   dbuffer_printf(buffer, "STRING \"");
428   while (*string) {
429     if (*string == '\n') {
430       string++;
431       continue;
432     }
433     if (*string == '"')
434       dbuffer_putc(buffer, '\\');
435     if (*string == ' ')
436       (*spaces)++;
437     dbuffer_putc(buffer, *string);
438     string++;
439   }
440   dbuffer_printf(buffer, "\"\n");
441 }
442
443 /* Using the _selected_ break gizmos, write _content_ to _output_. */
444 static void
445 print_gizmos(FILE *output, struct gizmo *gizmo, int line_width, int align)
446 {
447   int width, height, spaces;
448   struct style style;
449   struct dbuffer line;
450   struct dbuffer line_marks;
451   struct text_gizmo *text_gizmo;
452   struct break_gizmo *break_gizmo;
453   struct mark_gizmo *mark_gizmo;
454   width = 0;
455   height = 0;
456   spaces = 0;
457   style.font_name[0] = '\0';
458   style.font_size = 0;
459   /* _line_ is the text mode _pages_ content for this line. */
460   dbuffer_init(&line, 1024 * 4, 1024 * 4);
461   /* _line_marks_ are newline separated marks that appear on this line. */
462   dbuffer_init(&line_marks, 1024, 1024);
463   line_marks.data[0] = '\0';
464   for (; gizmo; gizmo = gizmo->next) {
465     switch (gizmo->type) {
466     case GIZMO_TEXT:
467       text_gizmo = (struct text_gizmo *)gizmo;
468       if (text_gizmo->style.font_size > height)
469         height = text_gizmo->style.font_size;
470       width += text_gizmo->width;
471       print_text(&line, &style, &text_gizmo->style, text_gizmo->string,
472           &spaces);
473       break;
474     case GIZMO_MARK:
475       mark_gizmo = (struct mark_gizmo *)gizmo;
476       dbuffer_printf(&line_marks, "^%s\n", mark_gizmo->string);
477       break;
478     case GIZMO_BREAK:
479       break_gizmo = (struct break_gizmo *)gizmo;
480       if (break_gizmo->style.font_size > height)
```

```
481         height = break_gizmo->style.font_size;
482       if (break_gizmo->selected) {
483         /* Line break occurs here so write and reset the line. */
484         width += break_gizmo->at_break_width;
485         print_text(&line, &style, &break_gizmo->style, break_gizmo->at_break,
486             &spaces);
487         /* If line not empty then write the line to _output_. */
488         if (line.size) {
489           dbuffer_putc(&line, '\0');
490           fprintf(output, "box %d\n", height);
491           /*
492            * Some align modes need the text to be horizontally shifted on the
493            * page. To do this, content enters graphic mode with START GRAPHIC
494            * and moves to the right with MOVE. Remember to end the graphic
495            * mode after exiting text mode.
496            */
497           if (align == ALIGN_CENTRE) {
498             fprintf(output, "START GRAPHIC\n");
499             fprintf(output, "MOVE %d 0\n", (line_width - width) / 2000);
500           } else if (align == ALIGN_RIGHT) {
501             fprintf(output, "START GRAPHIC\n");
502             fprintf(output, "MOVE %d 0\n", (line_width - width) / 1000);
503           }
504           fprintf(output, "START TEXT\n");
505           if (align == ALIGN_JUSTIFIED && spaces && !break_gizmo->force_break) {
506             fprintf(output, "SPACE %d\n", (line_width - width) / spaces);
507           }
508           fprintf(output, "%s", line.data);
509           fprintf(output, "END\n");
510           /* End the additional graphic mode. */
511           if (align == ALIGN_CENTRE || align == ALIGN_RIGHT) {
512             fprintf(output, "END\n");
513           }
514         }
515         fprintf(output, line_marks.data);
516         if (break_gizmo->spacing
517             && (break_gizmo->next == NULL || break_gizmo->next->next))
518           fprintf(output, "glue %d\n", break_gizmo->spacing);
519         fprintf(output, "opt_break\n");
520         /* Reset the line. */
521         line_marks.size = 0;
522         line_marks.data[0] = '\0';
523         height = 0;
524         width = 0;
525         spaces = 0;
526         style.font_name[0] = '\0';
527         style.font_size = 0;
528         line.size = 0;
529         line.data[0] = '\0';
530       } else {
531         width += break_gizmo->no_break_width;
532         print_text(&line, &style, &break_gizmo->style, break_gizmo->no_break,
533             &spaces);
534       }
535       break;
536     }
537   }
538   dbuffer_free(&line);
539   dbuffer_free(&line_marks);
540 }
541
542 static void
543 die_usage(char *program_name)
544 {
545   fprintf(stderr, "Usage: %s -w NUM [-l] [-r] [-j] [-c]\n", program_name);
546   exit(1);
547 }
548
549 int
550 main(int argc, char **argv)
551 {
552   int opt, line_width, align;
553   FILE *input_file, *typeface_file, *output_file;
554   struct typeface *typeface;
```

```
555   struct gizmo *gizmos;
556   line_width = 0;
557   align = ALIGN_LEFT;
558   while ( (opt = getopt(argc, argv, "ljrcw:")) != -1) {
559     switch (opt) {
560     case 'l':
561       align = ALIGN_LEFT;
562       break;
563     case 'j':
564       align = ALIGN_JUSTIFIED;
565       break;
566     case 'r':
567       align = ALIGN_RIGHT;
568       break;
569     case 'c':
570       align = ALIGN_CENTRE;
571       break;
572     case 'w':
573       if (str_to_int(optarg, &line_width))
574         die_usage(argv[0]);
575       /* Convert point space to text space. */
576       line_width *= 1000;
577       break;
578     default:
579       die_usage(argv[0]);
580     }
581   }
582   if (line_width == 0)
583     die_usage(argv[0]);
584   input_file = stdin;
585   typeface_file = fopen("typeface", "r");
586   if (typeface_file == NULL) {
587     fprintf(stderr, "Failed to open typeface file.\n");
588     return 1;
589   }
590   output_file = stdout;
591   typeface = open_typeface(typeface_file);
592   fclose(typeface_file);
593   gizmos = parse_gizmos(input_file, typeface);
594   optimise_breaks(gizmos, line_width);
595   print_gizmos(output_file, gizmos, line_width, align);
596   free_typeface(typeface);
597   free_gizmos(gizmos);
598   fclose(input_file);
599   fclose(output_file);
600   return 0;
601 }
```

dbuffer.c

```
001 /*
002  * Copyright (C) 2023 Christopher Lang
003  * See LICENSE for license details.
004  */
005
006 #include <stdio.h>
007 #include <stdarg.h>
008 #include <stdlib.h>
009
010 #include "tw.h"
011
012 void
013 dbuffer_init(struct dbuffer *buf, int initial, int increment)
014 {
015   buf->size = 0;
016   buf->allocated = initial;
017   buf->increment = increment;
018   buf->data = xmalloc(buf->allocated);
019 }
020
021 void
022 dbuffer_putc(struct dbuffer *buf, char c)
023 {
```

```
024   if (buf->allocated == buf->size) {
025     buf->allocated += buf->increment;
026     buf->data = xrealloc(buf->data, buf->allocated);
027   }
028   buf->data[buf->size++] = c;
029 }
030
031 void
032 dbuffer_printf(struct dbuffer *buf, const char *format, ...)
033 {
034   va_list args;
035   int len;
036   va_start(args, format);
037   /*
038    * Write to data but don't overflow the buffer. _len_ is the number of bytes
039    * that would have been written if the buffer were large enough.
040    */
041   len = vsnprintf(buf->data + buf->size, buf->allocated - buf->size, format,
042       args);
043   /* If there was not enough space allocated: */
044   if (len >= buf->allocated - buf->size) {
045     /* Allocate enough space for _len_ bytes. */
046     buf->allocated += buf->increment * (1 + len / buf->increment);
047     buf->data = xrealloc(buf->data, buf->allocated);
048     /*
049      * Write to data without checking overflow because we know there is enough
050      * space allocated.
051      */
052     vsprintf(buf->data + buf->size, format, args);
053   }
054   buf->size += len;
055   va_end(args);
056 }
057
058 void
059 dbuffer_free(struct dbuffer *buf)
060 {
061   free(buf->data);
062 }
```

record.c

```
001 /*
002  * Copyright (C) 2023 Christopher Lang
003  * See LICENSE for license details.
004  */
005
006 #include <stdio.h>
007 #include <stdlib.h>
008 #include <string.h>
009
010 #include "tw.h"
011
012 void
013 init_record(struct record *record)
014 {
015   dbuffer_init(&record->string, 1024 * 2, 1024 * 2);
016   record->field_count = 0;
017   record->fields_allocated = 32;
018   record->fields = xmalloc(record->fields_allocated * sizeof(char *));
019 }
020
021 void
022 begin_field(struct record *record)
023 {
024   if (record->field_count == record->fields_allocated) {
025     record->fields_allocated += 32;
026     record->fields = xrealloc(record->fields, record->fields_allocated);
027   }
028   record->fields[record->field_count++] = record->string.data
029     + record->string.size;
030 }
031
```

```
032 enum ParseState {
033   /* PARSING STATES */
034   PARSE_BEGIN,
035   PARSE_NORMAL_FIELD,
036   PARSE_NORMAL_FIELD_ESCAPE,
037   PARSE_QUOTED_FIELD,
038   PARSE_QUOTED_FIELD_ESCAPE,
039   PARSE_OUTSIDE_FIELD,
040   /* TERMINATING STATES */
041   PARSE_FINISH_RECORD,
042   PARSE_EOF,
043   /* ERROR STATES */
044   PARSE_UNTERMINATED_STRING,
045   PARSE_UNTERMINATED_ESCAPE,
046 };
047
048 int
049 parse_record(FILE *file, struct record *record)
050 {
051   int c, state;
052   state = PARSE_BEGIN;
053   record->string.size = 0;
054   record->field_count = 0;
055   /* State machine. */
056   while (state < PARSE_FINISH_RECORD) {
057     c = fgetc(file);
058     switch (state) {
059     case PARSE_BEGIN:
060       if (c == EOF) {
061         state = PARSE_EOF;
062       } else if (c == ' ' || c == '\n') {
063         continue;
064       } else if (c == '"') {
065         state = PARSE_QUOTED_FIELD;
066         begin_field(record);
067       } else if (c == '\\') {
068         state = PARSE_NORMAL_FIELD_ESCAPE;
069         begin_field(record);
070       } else {
071         state = PARSE_NORMAL_FIELD;
072         begin_field(record);
073         dbuffer_putc(&record->string, (char)c);
074       }
075       break;
076     case PARSE_NORMAL_FIELD:
077       if (c == EOF) {
078         state = PARSE_EOF;
079         dbuffer_putc(&record->string, '\0');
080       } else if (c == '\\') {
081         state = PARSE_NORMAL_FIELD_ESCAPE;
082       } else if (c == ' ') {
083         state = PARSE_OUTSIDE_FIELD;
084         dbuffer_putc(&record->string, '\0');
085       } else if (c == '\n') {
086         state = PARSE_FINISH_RECORD;
087         dbuffer_putc(&record->string, '\0');
088       } else {
089         dbuffer_putc(&record->string, (char)c);
090       }
091       break;
092     case PARSE_NORMAL_FIELD_ESCAPE:
093       if (c == EOF || c == '\n') {
094         state = PARSE_UNTERMINATED_ESCAPE;
095       } else {
096         state = PARSE_NORMAL_FIELD;
097         dbuffer_putc(&record->string, (char)c);
098       }
099       break;
100     case PARSE_QUOTED_FIELD:
101       if (c == EOF || c == '\n') {
102         state = PARSE_UNTERMINATED_STRING;
103       } else if (c == '\\') {
104         state = PARSE_QUOTED_FIELD_ESCAPE;
105       } else if (c == '"') {
```

```
106         state = PARSE_OUTSIDE_FIELD;
107         dbuffer_putc(&record->string, '\0');
108       } else {
109         dbuffer_putc(&record->string, (char)c);
110       }
111       break;
112     case PARSE_QUOTED_FIELD_ESCAPE:
113       if (c == EOF || c == '\n') {
114         state = PARSE_UNTERMINATED_ESCAPE;
115       } else {
116         state = PARSE_QUOTED_FIELD;
117         dbuffer_putc(&record->string, (char)c);
118       }
119       break;
120     case PARSE_OUTSIDE_FIELD:
121       if (c == EOF) {
122         state = PARSE_EOF;
123       } else if (c == '\n') {
124         state = PARSE_FINISH_RECORD;
125       } else if (c == ' ') {
126         continue;
127       } else if (c == '"') {
128         state = PARSE_QUOTED_FIELD;
129         begin_field(record);
130       } else if (c == '\\') {
131         state = PARSE_NORMAL_FIELD_ESCAPE;
132         begin_field(record);
133       } else {
134         state = PARSE_NORMAL_FIELD;
135         begin_field(record);
136         dbuffer_putc(&record->string, (char)c);
137       }
138       break;
139     default:
140       fprintf(stderr, "Unexpected state while parsing record.\n");
141       break;
142     }
143   }
144   /* _state_ is now in its final position. */
145   switch (state) {
146   case PARSE_UNTERMINATED_STRING:
147     fprintf(stderr, "Failed to parse record: unterminated string.\n");
148     goto error;
149   case PARSE_UNTERMINATED_ESCAPE:
150     fprintf(stderr, "Failed to parse record: unterminated escape sequence.\n");
151     goto error;
152   case PARSE_EOF:
153     return EOF;
154   case PARSE_FINISH_RECORD:
155     return 0;
156   default:
157     fprintf(stderr, "Unexpected terminating state while parsing record.\n");
158     goto error;
159   }
160 error:
161   record->field_count = 0;
162   record->string.size = 0;
163   return 1;
164 }
165
166 /* Find the index of field that matches _field_str_ otherwise return -1. */
167 int
168 find_field(const struct record *record, const char *field_str)
169 {
170   int i;
171   for (i = 0; i < record->field_count; i++)
172     if (strcmp(record->fields[i], field_str) == 0)
173       return i;
174   return -1;
175 }
176
177 void
178 free_record(struct record *record)
```

```
179 {
180   dbuffer_free(&record->string);
181   free(record->fields);
182 }
```

pdf.c

```
001 /*
002  * Copyright (C) 2023 Christopher Lang
003  * See LICENSE for license details.
004  */
005 
006 /*
007  * This file implements a PDF 1.7 writer. Get a copy of the 1.7 specification
008  * to understand the format.
009  */
010 
011 #include <stdio.h>
012 #include <string.h>
013 #include <stdlib.h>
014 #include <errno.h>
015 
016 #include "tw.h"
017 
018 static int pdf_add_font(FILE *pdf_file, FILE *font_file,
019     struct pdf_xref_table *xref, const char *name);
020 static int pdf_add_image(FILE *pdf_file, FILE *image_file,
021     struct pdf_xref_table *xref);
022 
023 void
024 pdf_write_header(FILE *file)
025 {
026   fprintf(file, "%%PDF-1.7\n");
027 }
028 
029 void
030 pdf_start_indirect_obj(FILE *file, struct pdf_xref_table *xref, int obj)
031 {
032   xref->obj_offsets[obj] = ftell(file);
033   fprintf(file, "%d 0 obj\n", obj);
034 }
035 
036 void
037 pdf_end_indirect_obj(FILE *file)
038 {
039   fprintf(file, "endobj\n");
040 }
041 
042 void
043 pdf_write_file_stream(FILE *pdf_file, FILE *data_file)
044 {
045   long size, i;
046   fseek(data_file, 0, SEEK_END);
047   size = ftell(data_file);
048   fseek(data_file, 0, SEEK_SET);
049   fprintf(pdf_file, "<<\n\
050   /Filter /ASCIIHexDecode\n\
051   /Length %ld\n\
052   /Length1 %ld\n\
053   >>\nstream\n", size * 2, size);
054   for (i = 0; i < size; i++)
055     fprintf(pdf_file, "%02x", (unsigned char)fgetc(data_file));
056   fprintf(pdf_file, "\nendstream\n");
057 }
058 
059 void
060 pdf_write_text_stream(FILE *file, const char *data, long size)
061 {
062   fprintf(file, "<< /Length %ld >> stream\n", size);
063   fwrite(data, 1, size, file);
064   fprintf(file, "\nendstream\n");
065 }
066 
```

```
067 void
068 pdf_write_int_array(FILE *file, const int *values, int count)
069 {
070   int i;
071   fprintf(file, "[\n ");
072   for (i = 0; i < count; i++)
073     fprintf(file, " %d", values[i]);
074   fprintf(file, "\n]\n");
075 }
076
077 void
078 pdf_write_font_descriptor(FILE *file, int font_file, const char *font_name,
079     int italic_angle, int ascent, int descent, int cap_height,
080     int stem_vertical, int min_x, int min_y, int max_x, int max_y)
081 {
082   fprintf(file, "<<\n\
083   /Type /FontDescriptor\n\
084   /FontName /%s\n\
085   /FontFile2 %d 0 R\n\
086   /Flags 6\n\
087   /FontBBox [%d, %d, %d, %d]\n\
088   /ItalicAngle %d\n\
089   /Ascent %d\n\
090   /Descent %d\n\
091   /CapHeight %d\n\
092   /StemV %d\n\
093 >>\n", font_name, font_file, min_x, min_y, max_x, max_y, italic_angle,
094       ascent, descent, cap_height, stem_vertical);
095 }
096
097 void
098 pdf_write_page(FILE *file, int parent, int content)
099 {
100   fprintf(file, "<<\n\
101   /Type /Page\n\
102   /Parent %d 0 R\n\
103   /Contents %d 0 R\n\
104 >>\n", parent, content);
105 }
106
107 void
108 pdf_write_font(FILE *file, const char *font_name, int font_descriptor,
109     int font_widths)
110 {
111   fprintf(file, "<<\n\
112   /Type /Font\n\
113   /Subtype /TrueType\n\
114   /BaseFont /%s\n\
115   /FirstChar 0\n\
116   /LastChar 255\n\
117   /Widths %d 0 R\n\
118   /FontDescriptor %d 0 R\n\
119 >>\n", font_name, font_widths, font_descriptor);
120 }
121
122 void
123 pdf_write_pages(FILE *file, int resources, int page_count, const int *page_objs)
124 {
125   int i;
126   fprintf(file, "<<\n\
127   /Type /Pages\n\
128   /Resources %d 0 R\n\
129   /Kids [\n", resources);
130   for (i = 0; i < page_count; i++)
131     fprintf(file, "    %d 0 R\n", page_objs[i]);
132   /* 595x842 is a portrait A4 page. */
133   fprintf(file, "    ]\n\
134   /Count %d\n\
135   /MediaBox [0 0 595 842]\n\
136 >>\n", page_count);
137 }
138
139 void
140 pdf_write_catalog(FILE *file, int page_list)
```

```
141 {
142   fprintf(file, "<<\n\
143   /Type /Catalog\n\
144   /Pages %d 0 R\n\
145 >>\n", page_list);
146 }
147
148 void
149 init_pdf_xref_table(struct pdf_xref_table *xref)
150 {
151   xref->obj_count = 1;
152   xref->allocated = 100;
153   xref->obj_offsets = xmalloc(xref->allocated * sizeof(long));
154   memset(xref->obj_offsets, 0, xref->allocated * sizeof(long));
155 }
156
157 int
158 allocate_pdf_obj(struct pdf_xref_table *xref)
159 {
160   if (xref->obj_count == xref->allocated) {
161     xref->obj_offsets
162       = xrealloc(xref->obj_offsets, (xref->allocated + 100) * sizeof(long));
163     memset(xref->obj_offsets + xref->allocated, 0, 100 * sizeof(long));
164     xref->allocated += 100;
165   }
166   return xref->obj_count++;
167 }
168
169 void
170 pdf_add_footer(FILE *file, const struct pdf_xref_table *xref, int root_obj)
171 {
172   int xref_offset, i;
173   xref_offset = ftell(file);
174   fprintf(file, "xref\n\
175 0 %d\n\
176 0000000000 65535 f \n", xref->obj_count);
177   for (i = 1; i < xref->obj_count; i++)
178     fprintf(file, "%09ld 00000 n \n", xref->obj_offsets[i]);
179   fprintf(file, "trailer << /Size %d /Root %d 0 R >>\n\
180 startxref\n\
181 %d\n\
182 %%%%EOF", xref->obj_count, root_obj, xref_offset);
183 }
184
185 void
186 free_pdf_xref_table(struct pdf_xref_table *xref)
187 {
188   free(xref->obj_offsets);
189 }
190
191 void
192 init_pdf_resources(struct pdf_resources *resources)
193 {
194   init_record(&resources->fonts_used);
195   init_record(&resources->images);
196 }
197
198 void
199 include_font_resource(struct pdf_resources *resources, const char *font)
200 {
201   int i;
202   for (i = 0; i < resources->fonts_used.field_count; i++)
203     if (strcmp(resources->fonts_used.fields[i], font) == 0)
204       return;
205   begin_field(&resources->fonts_used);
206   dbuffer_printf(&resources->fonts_used.string, "%s", font);
207   dbuffer_putc(&resources->fonts_used.string, '\0');
208 }
209
210 int
211 include_image_resource(struct pdf_resources *resources, const char *fname)
212 {
213   int i;
214   for (i = 0; i < resources->images.field_count; i++)
```

```
215     if (strcmp(resources->images.fields[i], fname) == 0)
216       return i;
217   begin_field(&resources->images);
218   dbuffer_printf(&resources->images.string, "%s", fname);
219   dbuffer_putc(&resources->images.string, '\0');
220   return i;
221 }
222
223 static int
224 pdf_add_font(FILE *pdf_file, FILE *font_file, struct pdf_xref_table *xref,
225     const char *name)
226 {
227   struct font_info font_info;
228   int font_program, font_widths, font_descriptor, font;
229
230   fseek(font_file, 0, SEEK_SET);
231   if (read_ttf(font_file, &font_info))
232     return -1;
233   fseek(font_file, 0, SEEK_SET);
234
235   font_program = allocate_pdf_obj(xref);
236   font_widths = allocate_pdf_obj(xref);
237   font_descriptor = allocate_pdf_obj(xref);
238   font = allocate_pdf_obj(xref);
239
240   pdf_start_indirect_obj(pdf_file, xref, font_program);
241   pdf_write_file_stream(pdf_file, font_file);
242   pdf_end_indirect_obj(pdf_file);
243
244   pdf_start_indirect_obj(pdf_file, xref, font_widths);
245   pdf_write_int_array(pdf_file, font_info.char_widths, 256);
246   pdf_end_indirect_obj(pdf_file);
247
248   pdf_start_indirect_obj(pdf_file, xref, font_descriptor);
249   pdf_write_font_descriptor(pdf_file, font_program, name, -10, 255,
250       255, 255, 10, font_info.x_min, font_info.y_min, font_info.x_max,
251       font_info.y_max);
252   pdf_end_indirect_obj(pdf_file);
253
254   pdf_start_indirect_obj(pdf_file, xref, font);
255   pdf_write_font(pdf_file, name, font_descriptor, font_widths);
256   pdf_end_indirect_obj(pdf_file);
257   return font;
258 }
259
260 static int
261 pdf_add_image(FILE *pdf_file, FILE *image_file, struct pdf_xref_table *xref)
262 {
263   int image;
264   long image_length, i;
265   struct jpeg_info jpeg_info;
266   const char *color_space;
267   if (read_jpeg(image_file, &jpeg_info))
268     return 1;
269   color_space = jpeg_info.components == 3 ? "DeviceRGB" : "DeviceGray";
270   fseek(image_file, 0, SEEK_END);
271   image_length = ftell(image_file);
272   fseek(image_file, 0, SEEK_SET);
273   image = allocate_pdf_obj(xref);
274   pdf_start_indirect_obj(pdf_file, xref, image);
275   fprintf(pdf_file, "<<\n\
276   /Type /XObject\n\
277   /Subtype /Image\n\
278   /Width %d\n\
279   /Height %d\n\
280   /ColorSpace /%s\n\
281   /BitsPerComponent 8\n\
282   /Length %ld\n\
283   /Filter /DCTDecode\n\
284   >>\n", jpeg_info.width, jpeg_info.height, color_space, image_length);
285   fprintf(pdf_file, "stream\n");
286   for (i = 0; i < image_length; i++)
287     fputc(fgetc(image_file), pdf_file);
288   fprintf(pdf_file, "\nendstream\n");
```

```
289   pdf_end_indirect_obj(pdf_file);
290   return image;
291 }
292
293 void
294 pdf_add_resources(FILE *pdf_file, FILE *typeface_file, int resources_obj,
295     const struct pdf_resources *resources, struct pdf_xref_table *xref)
296 {
297   FILE *font_file, *image_file;
298   struct record typeface_record;
299   int i, parse_result;
300   int *font_objs, *image_objs;
301   init_record(&typeface_record);
302   font_objs = xmalloc(resources->fonts_used.field_count * sizeof(int));
303   image_objs = xmalloc(resources->images.field_count * sizeof(int));
304   for (i = 0; i < resources->fonts_used.field_count; i++)
305     font_objs[i] = -1;
306   /* Create font resources for used fonts in the typeface. */
307   while ( (parse_result = parse_record(typeface_file, &typeface_record))
308       != EOF ) {
309     if (parse_result) /* If failed to parse record then skip it. */
310       continue;
311     if (typeface_record.field_count != 2) {
312       fprintf(stderr, "Typeface records must have exactly 2 fields.");
313       continue;
314     }
315     if (strlen(typeface_record.fields[0]) >= 256) {
316       fprintf(stderr,
317           "Typeface file contains font name that is too long '%s'.\n",
318           typeface_record.fields[0]);
319       continue;
320     }
321     if (!is_font_name_valid(typeface_record.fields[0])) {
322       fprintf(stderr, "Typeface file contains invalid font name '%s'.\n",
323           typeface_record.fields[0]);
324       continue;
325     }
326     /*
327      * If this font name is in _resources->fonts_used_ then it is used
328      * somewhere in the PDF so write the font resource to the PDF.
329      */
330     i = find_field(&resources->fonts_used, typeface_record.fields[0]);
331     if (i == -1) /* Font not used. */
332       continue;
333     /* Add the font resource. */
334     font_file = fopen(typeface_record.fields[1], "r");
335     if (font_file == NULL) {
336       fprintf(stderr, "Failed to open ttf file '%s': %s\n",
337           typeface_record.fields[1], strerror(errno));
338       continue;
339     }
340     font_objs[i] = pdf_add_font(pdf_file, font_file, xref,
341       typeface_record.fields[0]);
342     if (font_objs[i] == -1) {
343       fprintf(stderr, "Failed to parse ttf file '%s'\n",
344           typeface_record.fields[1]);
345     }
346     fclose(font_file);
347   }
348   free_record(&typeface_record);
349
350   /* Add image resources. */
351   for (i = 0; i < resources->images.field_count; i++) {
352     image_file = fopen(resources->images.fields[i], "r");
353     if (image_file == NULL) {
354       fprintf(stderr, "Failed to open image file '%s'\n",
355           resources->images.fields[i]);
356       continue;
357     }
358     image_objs[i] = pdf_add_image(pdf_file, image_file, xref);
359     fclose(image_file);
360   }
361
362   /* Add the PDF resources object. */
```

```
363   pdf_start_indirect_obj(pdf_file, xref, resources_obj);
364   fprintf(pdf_file, "<<\n  /Font <<\n");
365   for (i = 0; i < resources->fonts_used.field_count; i++) {
366     if (font_objs[i] == -1) {
367       fprintf(stderr, "Typeface file does not include '%s' font.\n",
368           resources->fonts_used.fields[i]);
369     }
370     fprintf(pdf_file, "    /%s %d 0 R\n", resources->fonts_used.fields[i],
371         font_objs[i]);
372   }
373   fprintf(pdf_file, "  >>\n  /XObject <<\n");
374   for (i = 0; i < resources->images.field_count; i++)
375     fprintf(pdf_file, "    /Img%d %d 0 R\n", i, image_objs[i]);
376   fprintf(pdf_file, "  >>\n>>\n");
377   pdf_end_indirect_obj(pdf_file);
378   free(font_objs);
379   free(image_objs);
380 }
381
382 void
383 free_pdf_resources(struct pdf_resources *resources)
384 {
385   free_record(&resources->fonts_used);
386   free_record(&resources->images);
387 }
388
389 void
390 init_pdf_page_list(struct pdf_page_list *page_list)
391 {
392   page_list->page_count = 0;
393   page_list->pages_allocated = 100;
394   page_list->page_objs = xmalloc(page_list->pages_allocated * sizeof(int));
395 }
396
397 void
398 add_pdf_page(struct pdf_page_list *page_list, int page)
399 {
400   if (page_list->page_count == page_list->pages_allocated) {
401     page_list->pages_allocated += 100;
402     page_list->page_objs = xrealloc(page_list->page_objs,
403         page_list->pages_allocated);
404   }
405   page_list->page_objs[page_list->page_count++] = page;
406 }
407
408 void
409 free_pdf_page_list(struct pdf_page_list *page_list)
410 {
411   free(page_list->page_objs);
412 }
```

jpeg.c

```
001 /*
002  * Copyright (C) 2023 Christopher Lang
003  * See LICENSE for license details.
004  */
005
006 /*
007  * This file parses JPEG files. See 'https://en.wikipedia.org/wiki/JPEG' for
008  * file format information.
009  */
010
011 #include <stdio.h>
012 #include <stdint.h>
013 #include "tw.h"
014
015 static uint16_t read_uint16(FILE *file);
016 static void skip_segment_payload(FILE *file);
017
018 /* Read big-endian 16-bit unsigned integer from file. */
019 static uint16_t
020 read_uint16(FILE *file)
```

```
021 {
022   uint16_t n;
023   n = 0;
024   fread(1 + (char *)&n, 1, 1, file);
025   fread((char *)&n, 1, 1, file);
026   return n;
027 }
028
029 static void
030 skip_segment_payload(FILE *file)
031 {
032   uint16_t segment_length;
033   segment_length = read_uint16(file);
034   fseek(file, segment_length - 2, SEEK_CUR);
035 }
036
037 int
038 read_jpeg(FILE *file, struct jpeg_info *info)
039 {
040   unsigned char magic_bytes[2];
041   unsigned char segment_code[2];
042
043   fread(magic_bytes, 1, 2, file);
044   if (magic_bytes[0] != 0xff || magic_bytes[1] != 0xd8) {
045     fprintf(stderr, "File not JPEG.\n");
046     return 1;
047   }
048
049   for(;;) {
050     fread(segment_code, 1, 2, file);
051     if (segment_code[0] != 0xFF) {
052       fprintf(stderr, "JPEG file invalid.\n");
053       return 1;
054     }
055     if (segment_code[1] >= 0xe0 && segment_code[1] <= 0xef) {
056       /*
057        * Application specific segment.
058        * Just hope that the first two bytes specify length of this segment.
059        */
060       skip_segment_payload(file);
061     } else {
062       switch (segment_code[1]) {
063       case 0xfe: /* comment marker */
064       case 0xdb: /* quantization tables */
065       case 0xc4: /* huffman tables */
066         skip_segment_payload(file);
067         break;
068       case 0xc0: /* baseline DCT segment (contains image width and height) */
069         fseek(file, 3, SEEK_CUR);
070         info->height = read_uint16(file);
071         info->width = read_uint16(file);
072         fread(&info->components, 1, 1, file);
073         goto end_scan;
074       case 0xc2: /* progressive DCT segment (unsupported) */
075         fprintf(stderr, "Progressive DCT JPEGs are unsupported.\n");
076         return 1;
077       case 0xda: /* compressed image data */
078         /*
079          * If we get to the image data and have not found width and height yet
080          * then assume we will not find it.
081          */
082         fprintf(stderr, "JPEG image data reached and no DCT segment found.\n");
083         return 1;
084       case 0xd9: /* end of image marker */
085         fprintf(stderr, "End of JPEG reached and no DCT segment found.\n");
086         return 1;
087       default:
088         fprintf(stderr, "Unrecognised JPEG segment code '%02x %02x'\n",
089             segment_code[0], segment_code[1]);
090         return 1;
091       }
092     }
093     if (ferror(file) || feof(file)) {
094       fprintf(stderr, "Error reading JPEG file.\n");
```

```
095       return 1;
096     }
097   }
098 end_scan:
099   if (ferror(file) || feof(file)) {
100     fprintf(stderr, "Error reading JPEG file.\n");
101     return 1;
102   }
103   if (info->components != 1 && info->components != 3) {
104     fprintf(stderr, "JPEG file has unsupported color component count '%d'.\n",
105         info->components);
106     return 1;
107   }
108   return 0;
109 }
```

ttf.c

```
001 /*
002  * Copyright (C) 2023 Christopher Lang
003  * See LICENSE for license details.
004  */
005
006 /*
007  * See Apple's TrueType reference manual for file format info:
008  * https://developer.apple.com/fonts/TrueType-Reference-Manual/
009  */
010
011 #include <stdint.h>
012 #include <stdio.h>
013 #include <string.h>
014 #include <stdlib.h>
015
016 #include "tw.h"
017
018 #define REQUIRED_TABLE_COUNT \
019   (sizeof(required_tables) / sizeof(required_tables[0]))
020
021 static int16_t read_int16(const char *ptr);
022 static int32_t read_int32(const char *ptr);
023 static int read_head_table(
024     const char *table, long table_size, struct font_info *info);
025 static int read_format4_cmap_subtable(
026     const char *table, long table_size, struct font_info *info);
027 static int read_cmap_table(
028     const char *table, long table_size, struct font_info *info);
029 static int read_hhea_table(
030     const char *table, long table_size, struct font_info *info);
031 static int read_hmtx_table(
032     const char *table, long table_size, struct font_info *info);
033
034 static const char *required_tables[] = {"head", "cmap", "hhea", "hmtx"};
035 static int (*required_table_parsers[REQUIRED_TABLE_COUNT])
036   (const char *table, long table_size, struct font_info *font) = {
037   read_head_table,
038   read_cmap_table,
039   read_hhea_table,
040   read_hmtx_table,
041 };
042
043 /* Convert big-edian to little-endian. Assume this machine is little-endian. */
044 static int16_t
045 read_int16(const char *ptr)
046 {
047   uint16_t ret;
048   ((char *)&ret)[0] = ptr[1];
049   ((char *)&ret)[1] = ptr[0];
050   return ret;
051 }
052
053 static int32_t
054 read_int32(const char *ptr)
055 {
```

```
056   uint32_t ret;
057   ((char *)&ret)[0] = ptr[3];
058   ((char *)&ret)[1] = ptr[2];
059   ((char *)&ret)[2] = ptr[1];
060   ((char *)&ret)[3] = ptr[0];
061   return ret;
062 }
063
064 static int
065 read_head_table(const char *table, long table_size, struct font_info *info)
066 {
067   if (table_size != 54)
068     return 1;
069   info->units_per_em = read_int16(table + 18);
070   info->x_min = read_int16(table + 36) * 1000 / info->units_per_em;
071   info->y_min = read_int16(table + 38) * 1000 / info->units_per_em;
072   info->x_max = read_int16(table + 40) * 1000 / info->units_per_em;
073   info->y_max = read_int16(table + 42) * 1000 / info->units_per_em;
074   return 0;
075 }
076
077 static int
078 read_format4_cmap_subtable(
079     const char *table,
080     long table_size,
081     struct font_info *info)
082 {
083   unsigned int range_index, char_index;
084   uint16_t seg_count;
085   const char *end_codes, *start_codes, *deltas, *offsets;
086   if (table_size < 16)
087     return 1;
088   seg_count = read_int16(table + 6) / 2;
089   end_codes = table + 14;
090   start_codes = end_codes + seg_count * 2 + 2;
091   deltas = start_codes + seg_count * 2;
092   offsets = deltas + seg_count * 2;
093   if (table_size < 16 + seg_count * 8)
094     return 1;
095   for (char_index = 0; char_index < 256; char_index++)
096     info->cmap[char_index] = 0;
097   for (range_index = 0; range_index < seg_count; range_index++) {
098     uint16_t start, end, glyph_delta, glyph_offset;
099     end = read_int16(end_codes + range_index * 2);
100     start = read_int16(start_codes + range_index * 2);
101     if (end > 255) end = 255;
102     if (start > 255) break;
103     glyph_delta = read_int16(deltas + range_index * 2);
104     glyph_offset = read_int16(offsets + range_index * 2);
105     if (glyph_offset == 0) {
106       for (char_index = start; char_index <= end; char_index++)
107         info->cmap[char_index] = char_index + glyph_delta;
108     } else {
109       if(offsets + range_index * 2 + glyph_offset
110           + 2 * (end - start) + 2
111           > table + table_size)
112         return 1;
113       for (char_index = start; char_index <= end; char_index++)
114         info->cmap[char_index] = read_int16(
115           offsets + range_index * 2
116           + glyph_offset
117           + 2 * (char_index - start));
118     }
119   }
120   return 0;
121 }
122
123 static int
124 read_cmap_table(const char *table, long table_size, struct font_info *info)
125 {
126   uint16_t subtable_count, format, subtable_size;
127   const char *subtable;
128   if (table_size < 4)
129     goto invalid;
```

```
130   subtable_count = read_int16(table + 2);
131   if (table_size < 4 + subtable_count * 8)
132     goto invalid;
133   for (subtable = table + 4;
134       subtable < table + 4 + subtable_count * 8;
135       subtable += 8) {
136     uint16_t platform_id, platform_specific_id;
137     uint32_t offset;
138     platform_id = read_int16(subtable);
139     platform_specific_id = read_int16(subtable + 2);
140     offset = read_int32(subtable + 4);
141     if (platform_id == 0 && platform_specific_id <= 4) {
142       subtable = table + offset;
143       goto found_cmap;
144     }
145   }
146   goto unsupported;
147 found_cmap:
148   if (subtable + 4 > table + table_size)
149     goto invalid;
150   format = read_int16(subtable);
151   subtable_size = read_int16(subtable + 2);
152   if (subtable + subtable_size > table + table_size)
153     goto invalid;
154   if (format != 4)
155     goto unsupported;
156   if (read_format4_cmap_subtable(subtable, subtable_size, info))
157     goto subtable_error;
158   return 0;
159 invalid:
160   fprintf(stderr, "ttf cmap table invalid\n");
161   return 1;
162 unsupported:
163   fprintf(stderr, "ttf cmap table not supported\n");
164   return 1;
165 subtable_error:
166   fprintf(stderr, "failed to read ttf cmap subtable\n");
167   return 1;
168 }
169
170 static int
171 read_hhea_table(const char *table, long table_size, struct font_info *info)
172 {
173   if (table_size != 36)
174     return 1;
175   info->long_hor_metrics_count = read_int16(table + 34);
176   return 0;
177 }
178
179 static int
180 read_hmtx_table(const char *table, long table_size, struct font_info *info)
181 {
182   int i;
183   uint16_t fallback_tt_width;
184   if (info->long_hor_metrics_count < 1)
185     return 1;
186   if (table_size < info->long_hor_metrics_count * 4)
187     return 1;
188   fallback_tt_width
189     = read_int16(table + 4 * (info->long_hor_metrics_count - 1));
190   for (i = 0; i < 256; i++) {
191     uint16_t glyph, tt_width;
192     glyph = info->cmap[i];
193     tt_width
194       = glyph >= info->long_hor_metrics_count
195       ? fallback_tt_width
196       : read_int16(table + 4 * glyph);
197     info->char_widths[i] = tt_width * 1000 / info->units_per_em;
198   }
199   return 0;
200 }
201
202 int
203 read_ttf(FILE *file, struct font_info *info)
```

```
204 {
205   /*
206    * This file was originally written to parse the ttf data from a buffer. I
207    * later decided to change the function definition to read straight from a
208    * file descriptor. I couldn't be bothered to rewrite all the ttf parsing
209    * code to read right from the file, so I added the following hack that reads
210    * the file into a sufficiently large buffer and then the rest of the code
211    * works as it did before.
212    */
213   char *ttf;
214   long ttf_size;
215   fseek(file, 0, SEEK_END);
216   ttf_size = ftell(file);
217   fseek(file, 0, SEEK_SET);
218   ttf = xmalloc(ttf_size);
219   fread(ttf, 1, ttf_size, file);
220
221   int i;
222   uint32_t magic_bytes;
223   uint16_t table_count;
224   uint32_t required_table_offsets[REQUIRED_TABLE_COUNT];
225   uint32_t required_table_lengths[REQUIRED_TABLE_COUNT];
226   const char *table_directory;
227   const char *table;
228
229   if (ttf_size < 12)
230     goto table_directory_invalid;
231   magic_bytes = read_int32(ttf);
232   if (magic_bytes != 0x00010000)
233     goto not_ttf;
234   table_count = read_int16(ttf + 4);
235   table_directory = ttf + 12;
236   if (ttf_size < 12 + table_count * 16)
237     goto table_directory_invalid;
238   for (i = 0; i < REQUIRED_TABLE_COUNT; i++)
239     required_table_lengths[i] = 0;
240   for (table = table_directory;
241       table < table_directory + table_count * 16;
242       table += 16) {
243     char tag[4];
244     uint32_t offset, length;
245     memcpy(tag, table, 4);
246     offset = read_int32(table + 8);
247     length = read_int32(table + 12);
248     for (i = 0; i < REQUIRED_TABLE_COUNT; i++)
249       if (strcmp(tag, required_tables[i]) == 0) {
250         required_table_offsets[i] = offset;
251         required_table_lengths[i] = length;
252       }
253   }
254   for (i = 0; i < REQUIRED_TABLE_COUNT; i++) {
255     if (required_table_lengths[i] == 0)
256       goto table_no_exist;
257     if (required_table_parsers[i](ttf + required_table_offsets[i],
258           required_table_lengths[i], info))
259       goto table_error;
260   }
261   free(ttf);
262   return 0;
263 not_ttf:
264   fprintf(stderr, "failed to read ttf: not a ttf\n");
265   free(ttf);
266   return 1;
267 table_directory_invalid:
268   fprintf(stderr, "ttf table directory invalid\n");
269   free(ttf);
270   return 1;
271 table_no_exist:
272   fprintf(stderr, "ttf does not have %s table\n", required_tables[i]);
273   free(ttf);
274   return 1;
275 table_error:
```

```
276   fprintf(stderr, "failed to read ttf %s table\n", required_tables[i]);
277   free(ttf);
278   return 1;
279 }
```

utils.c

```
001 /*
002  * Copyright (C) 2023 Christopher Lang
003  * See LICENSE for license details.
004  */
005
006 #include <stdlib.h>
007 #include <stdio.h>
008
009 #include "tw.h"
010
011 /* Call _malloc_ and catch memory error. */
012 void *
013 xmalloc(size_t len)
014 {
015   void *p;
016   if ( (p = malloc(len)) == NULL)
017     perror("malloc");
018   return p;
019 }
020
021 /* Call _realloc_ and catch memory error. */
022 void *
023 xrealloc(void *p, size_t len)
024 {
025   if ( (p = realloc(p, len)) == NULL)
026     perror("realloc");
027   return p;
028 }
029
030 int
031 is_font_name_valid(const char *font_name)
032 {
033   unsigned char c;
034   if (*font_name == '\0')
035     return 1;
036   while (*font_name) {
037     c = *font_name;
038     /* '-' or 0-9 or '_' or A-Z or a-z */
039     if (c == 45 || (c >= 48 && c <= 57) || c == 95 || (c >= 65 && c <= 90)
040         || (c >= 97 && c <= 122)) {
041       font_name++;
042     } else {
043       return 0;
044     }
045   }
046   return 1;
047 }
048
049 int
050 str_to_int(const char *str, int *n)
051 {
052   char *endptr;
053   *n = strtol(str, &endptr, 10);
054   if (*str == '\0' || *endptr != '\0')
055     return 1;
056   return 0;
057 }
```

utils.py

```
001 import sys, subprocess, re
002
003 # utils.py
004 # Provides utility functions for python scripts part of the typesetting
005 # pipeline.
006
007 # This function is used to invoke the _line_break_ binary. _text_ is passed to
008 # the _line_break_ program's standard input and this function returns the
009 # output when its finished. We assume the _line_break_ binary is in the users
010 # PATH environment variable.
011 def line_break(text, width, align):
012   # Invoke the _line_break_ program with command line arguments.
013   process = subprocess.Popen(["line_break", "-" + align, "-w", str(width)],
014                              stdin=subprocess.PIPE,
015                              stdout=subprocess.PIPE)
016   # Encode the text, write it to the process's standard input stream.
017   text = bytes(text, "ascii")
018   process.stdin.write(text)
019   # Wait for the program to exit and get the standard output data.
020   output, error = process.communicate()
021   # Decode and return the standard output.
022   return output.decode("ascii")
023
024 # Print a string to standard error stream.
025 def warn(msg):
026   print(msg, file=sys.stderr)
027
028 # Prepare a string to be put in a quoted record field.
029 # The backslash is used to escape characters in the field.
030 # Backslash literals need to be escaped with another backslash.
031 # A quotation mark is used to end the field. A quotation mark literal must be
032 # escaped with a backslash.
033 # Newlines are not allowed.
034 def strip_string(string):
035   return string.replace('\\', '\\\\').replace('"', '\\"').replace('\n', '')
036
037 # Parse the next record that appears in _file_.
038 def parse_record(file):
039   fields = []
040   # Try to parse a record on each line until a record is found.
041   for line in file:
042     fields = re.findall(r'[^"\s]\S*|".*?[^\\]"', line)
043     if len(fields):
044       break
045   # If the end of the file was reached without finding a record: return None.
046   if len(fields) == 0:
047     return None
048   # Remove quotes from quoted fields; remove backslash from quotation mark
049   # literal.
050   for i in range(len(fields)):
051     if fields[i][0] == '"':
052       fields[i] = fields[i][1:-1] # Remove first and last character (").
053     fields[i] = fields[i].replace('\\"', '"')
054   return fields
```

contents.py

```
001 #!/bin/python3
002
003 # contents.py
004 # Generates contents page's content.
005 # Parses standard input in _contents_ format and writes _content_ to standard
006 # output.
007
008 import sys, argparse
009 from utils import *
010
011 # Parse command line arguments.
012 arg_parser = argparse.ArgumentParser()
013 arg_parser.add_argument("-c", "--char_width", type=int, default=60)
```

```
014 arg_parser.add_argument("-s", "--font_size", type=int, default=12)
015 args = arg_parser.parse_args()
016
017 char_width = args.char_width
018 font_size = args.font_size
019
020 # Keep parsing records from standard input until reach end.
021 # fields is an array of fields in this record.
022 while fields := parse_record(sys.stdin):
023   if len(fields) != 2:
024     warn("contents file record must have 2 fields")
025     continue
026   # _padding_ is how many dots we need to separate the section name from the
027   # page number.
028   padding = char_width - len(fields[0]) - len(fields[1])
029   # If the text overflows the width then don't add any dots.
030   if padding < 0:
031     padding = 0
032   # Build the _content_ text for this line.
033   # A monospaced font is used to make dots align in each line of
034   # the contents page.
035   graphic = "box {}\n".format(args.font_size)
036   graphic += "START TEXT\n"
037   graphic += "FONT Monospace {}\n".format(args.font_size)
038   string = fields[0] + '.' * padding + fields[1]
039   # _strip_string_ removes illegal characters from the string.
040   graphic += 'STRING "{}"\n'.format(strip_string(string))
041   graphic += "END\n"
042   # A page break may occur in-between contents lines.
043   graphic += "opt_break\n"
044   # Write this line's _content_ to standard output.
045   print(graphic, end='')
```

markup_raw.py

```
001 #!/bin/python3
002
003 # markup_raw.py
004 # Read text from standard input, preserve line breaks and write _content_ text
005 # to standard output with optional breaks inbetween lines.
006
007 import sys, argparse
008 from utils import *
009
010 # Orphans are isolated lines of text at the bottom of a page.
011 # Widows are isolated lines of text at the top of a page.
012
013 arg_parser = argparse.ArgumentParser()
014 arg_parser.add_argument("-s", "--font_size", type=int, default=12)
015 arg_parser.add_argument("-f", "--font_name", default="Monospace")
016 arg_parser.add_argument("-o", "--orphans", type=int, default=1)
017 arg_parser.add_argument("-w", "--widows", type=int, default=1)
018 args = arg_parser.parse_args()
019
020 font_size = args.font_size
021 font = args.font_name
022 # _orphans_ is the maximum number of allowed orphans.
023 orphans = args.orphans
024 # _widows_ is the maximum number of allowed widowsd.
025 widows = args.widows
026
027 # Read all lines from standard input and loop over them.
028 lines = sys.stdin.readlines()
029 i = 0
030 for line in lines:
031   i += 1
032   # Make a box for this line with the line text in it and write this to stdout.
033   print("box {}".format(font_size))
034   print("START TEXT")
035   print("FONT {} {}".format(strip_string(font), font_size))
036   print('STRING "{}"'.format(strip_string(line)))
037   print("END")
038   # Only allow a page break here if it does not result in too many orphans or
```

```
039   # widows.
040   if i >= orphans and len(lines) - i >= widows:
041     print("opt_break")
042 # Allow a page break after all lines of text.
043 print("opt_break")
```

markup_text.py

```
001 #!/bin/python3
002
003 # markup_text.py
004 # Read markup text from standard input.
005 # Typeset this text and write _content_ to standard output.
006
007 import sys, argparse
008 from utils import *
009
010 # A _TextStream_ builds a _text_specification_.
011 # By invoking _line_break_ this can be converted into _content_.
012 class TextStream:
013   def __init__(self, width, align, paragraph_spacing, line_spacing):
014     self.width = width
015     # _align_ is the align mode for this text. 'l', 'r', 'c' or 'j'.
016     self.align = align
017     self.paragraph_spacing = paragraph_spacing
018     self.line_spacing = line_spacing
019     self.in_paragraph = False
020     self.in_string = False
021     # _text_ is the _text_specification_ string.
022     self.text = ""
023     # An insertion is some _content_ to be inserted after the line of text that
024     # the insertion appears on. For example, a footnote is a type of insertion.
025     self.insertions = []
026   # Add a string to the _text_specification_.
027   def add_string(self, string):
028     # If not already in a string then start one.
029     if not self.in_string:
030       self.text += 'STRING "'
031       self.in_string = True
032     # Add the stripped text to the _text_specification_.
033     self.text += strip_string(string)
034   def close_string(self):
035     # If _text_specification_ is in a string then close it and get ready for
036     # the next _text_specification_ command.
037     if self.in_string:
038       self.text += '"\n'
039       self.in_string = False
040   # Sets the font to be used for subsequent strings.
041   def set_font(self, font_name, font_size):
042     # Close the string if we are in one.
043     self.close_string()
044     # Add the FONT command to the _text_specification_.
045     self.text += "FONT {} {}\n".format(font_name, font_size)
046   # Add a word to the _text_specification_.
047   def add_word(self, word):
048     # If empty do nothing.
049     if len(word) == 0:
050       return
051     # Close a string if one is open.
052     self.close_string()
053     # If this is not the first word of the paragraph add an optional break
054     # that inserts a space when no break occurs.
055     if self.in_paragraph:
056       self.text += 'OPTBREAK " " "" {}\n'.format(self.line_spacing)
057     # We are now in a paragraph
058     self.in_paragraph = True
059     # The word itself is added.
060     self.add_string(word)
061   def end_paragraph(self):
062     if self.in_paragraph:
063       self.close_string()
064       # Add a forced line break here.
065       self.text += "BREAK {}\n".format(self.paragraph_spacing)
```

```
066       self.in_paragraph = False
067     # _insertion_ is _content_ that must appear on this line.
068     def insert_content(self, insertion):
069       self.close_string()
070       # Get the unique identifier for this mark.
071       mark = len(self.insertions)
072       # Add the insertion to the list of insertions.
073       self.insertions.append(insertion)
074       # Add the MARK into the text_specification.
075       self.text += "MARK {}\n".format(mark)
076     # Convert this _text_specification_ to _content_ by finding optimal line
077     # breaks.
078     def to_content(self):
079       self.close_string()
080       # Invoke the line_break binary with _self.text_ input.
081       content = line_break(self.text, self.width, self.align)
082       # Insert the insertion content after line breaking.
083       for i in range(len(self.insertions)):
084         content = content.replace('\n^' + str(i), '\n' + self.insertions[i])
085       return content
086     # Read whitespace separated words from line into the text stream.
087     def read_words(self, line):
088       # Split the line into words by whitespace.
089       words = line.split()
090       # Add each word individually.
091       for word in words:
092         self.add_word(word)
093
094 # The _MainStream_ INHERITS from text _TextStream_.
095 # It is used for passing all markup text excluding the content of footnotes.
096 # It is able to parse bold text, footnotes and other markup features.
097 class MainStream(TextStream):
098     def __init__(self, normal_width, footnote_width, normal_size, footnote_size, \
099         normal_align, footnote_align, normal_paragraph_spacing, \
100         normal_line_spacing, footnote_paragraph_spacing, footnote_line_spacing):
101       # Initialise the superclass.
102       super().__init__(normal_width, normal_align, normal_paragraph_spacing, \
103           normal_line_spacing)
104       self.footnote_width = footnote_width
105       self.normal_size = normal_size
106       self.footnote_size = footnote_size
107       self.footnote_align = footnote_align
108       self.footnote_line_spacing = footnote_line_spacing
109       self.footnote_paragraph_spacing = footnote_paragraph_spacing
110       self.set_font("Regular", self.normal_size)
111       # Remember the last font set so that if more Regular text is encountered
112       # there is no need to unnecessarily set the font again.
113       self.font_mode = 'R'
114     # Add a footnote to the stream.
115     def read_footnote(self, footnote_symbol, footnote_text):
116       # Make a new stream for this footnotes content.
117       footnote_stream = TextStream(self.footnote_width, self.footnote_align, \
118           self.footnote_paragraph_spacing, self.footnote_line_spacing)
119       # Write footnote symbol and text to the new stream.
120       footnote_stream.set_font("Regular", self.footnote_size)
121       footnote_stream.add_word(footnote_symbol)
122       footnote_stream.set_font("Italic", self.footnote_size)
123       footnote_stream.read_words(footnote_text)
124       # Convert the stream to content (line break the footnote).
125       content = footnote_stream.to_content()
126       # Add some glue to the end so footnotes are separated.
127       content += "glue {}\n".format(self.footnote_paragraph_spacing)
128       # Set the flow for the footnote content.
129       content = "flow footnote\n" + content + "flow normal\n"
130       # Insert the footnote content into the MainStream at this point.
131       self.insert_content(content)
132     # Read a line of markup text and add it to the stream.
133     def read_line(self, line):
134       # If the line begins with a carrot (^) then it is a footnote.
135       if line[0] == '^':
136         # The first part of the footnote after the carrot (&) is the footnote
137         # symbol. The rest is the footnote text.
138         parts = line[1:].split(maxsplit = 1)
139         # If there was not a footnote synbol and text after the carrot then
```

```
140       # ignore this line.
141       if len(parts) < 2:
142         return
143       footnote_symbol, footnote_text = parts
144       # Write the footnote symbol to main text and add the footnote itself.
145       self.add_word(footnote_symbol)
146       self.read_footnote(footnote_symbol, footnote_text)
147       # Exit the function.
148       return
149     # A line starting with one or more hashtags (#) is a header.
150     if line[0] == '#':
151       # Remove the first hashtag.
152       line = line[1:]
153       level = 1
154       # For each subsequent hashtag, increase the level by 1.
155       while line[0] == '#':
156         line = line[1:]
157         level += 1
158       # More than 2 hashtags make no difference.
159       if level > 2:
160         level = 2
161       # Calculate the text size for the header.
162       size = int(self.normal_size * 1.62 ** (3 - level))
163       # End any open paragraphs.
164       self.end_paragraph()
165       # Write the header with _size_.
166       self.set_font("Regular", size)
167       self.read_words(line)
168       # Return to normal size and font.
169       self.set_font("Regular", self.normal_size)
170       # The header is technically a paragraph that needs to be ended.
171       self.end_paragraph()
172       self.font_mode = 'R'
173       # Exit the function.
174       return
175     # The line is a list of whitespace separated words.
176     words = line.split()
177     # Loop through each word.
178     for word in words:
179       # Bold text is enclosed in stars (*), italic in underscores (_).
180       # If the word begins with a star (*) or underscore (_) and the font mode
181       # is Regular. Then we switch to the new font mode and remove the star or
182       # underscore from the start of the word.
183       if word[0] == '*' and self.font_mode == 'R':
184         self.set_font("Bold", self.normal_size)
185         word = word[1:]
186         self.font_mode = 'B'
187       elif word[0] == '_' and self.font_mode == 'R':
188         self.set_font("Italic", self.normal_size)
189         word = word[1:]
190         self.font_mode = 'I'
191       # If the word is empty then go to the next word.
192       if len(word) == 0:
193         continue
194       # If a star (*) or underscore (_) ends bold or italic font mode. Then
195       # remove the star or underscore from the end of the word, add the word
196       # and return to Regular font mode.
197       # Else: if its just a normal word then add it.
198       if word[-1] == '*' and self.font_mode == 'B':
199         word = word[:-1]
200         self.add_word(word)
201         self.set_font("Regular", self.normal_size)
202         self.font_mode = 'R'
203       elif word[-1] == '_' and self.font_mode == 'I':
204         word = word[:-1]
205         self.add_word(word)
206         self.set_font("Regular", self.normal_size)
207         self.font_mode = 'R'
208       else:
209         self.add_word(word)
210     # If there were no words on this line then end the paragraph.
211     if len(words) == 0:
212       self.end_paragraph()
213
```

```
214 # Parse command line arguments.
215 arg_parser = argparse.ArgumentParser()
216 arg_parser.add_argument("-w", "--normal_width", type=int, required=True)
217 arg_parser.add_argument("-W", "--footnote_width", type=int)
218 arg_parser.add_argument("-s", "--normal_size", type=int, default=12)
219 arg_parser.add_argument("-S", "--footnote_size", type=int, default=12)
220 arg_parser.add_argument("-a", "--normal_align", default='l')
221 arg_parser.add_argument("-A", "--footnote_align", default='l')
222 arg_parser.add_argument("-l", "--normal_line_spacing", default=0)
223 arg_parser.add_argument("-L", "--footnote_line_spacing", default=0)
224 arg_parser.add_argument("-p", "--normal_paragraph_spacing")
225 arg_parser.add_argument("-P", "--footnote_paragraph_spacing")
226 args = arg_parser.parse_args()
227
228 # Some command line arguments default to values of other arguments.
229 if args.footnote_width == None:
230   args.footnote_width = args.normal_width
231 if args.normal_paragraph_spacing == None:
232   args.normal_paragraph_spacing = args.normal_size
233 if args.footnote_paragraph_spacing == None:
234   args.footnote_paragraph_spacing = args.footnote_size
235
236 # Verify align mode command line arguments are valid.
237 # 'l', 'r', 'c', 'j' are text alignment modes, short for: left, right, centre,
238 # justified.
239 if not args.normal_align in ('l', 'r', 'c', 'j'):
240   warn("Invalid normal align mode.")
241   exit(1)
242 if not args.normal_align in ('l', 'r', 'c', 'j'):
243   warn("Invalid footnote align mode.")
244   exit(1)
245
246 # Create a MainStream with the command line options.
247 main_stream = MainStream(args.normal_width, args.footnote_width, \
248     args.normal_size, args.footnote_size, args.normal_align, \
249     args.footnote_align, args.normal_paragraph_spacing, \
250     args.normal_line_spacing, args.footnote_paragraph_spacing, \
251     args.footnote_line_spacing)
252 # For each line in the standard input, parse this line with the MainStream.
253 for line in sys.stdin:
254   main_stream.read_line(line)
255 # Convert the MainStream to _content_ and write to standard output.
256 print(main_stream.to_content(), end='')
```

pager.py

```
001 #!/bin/python3
002
003 # pager.py
004 # Read _content_ from standard input, split into _pages_ which are written to
005 # standard output.
006
007 import sys, argparse, re
008 from utils import *
009
010 # Return next non-empty line from _file_.
011 def next_line(file):
012   # Loop until a good line is found.
013   while True:
014     line = file.readline()
015     # If we reach the end of the file then return None.
016     if not line:
017       return None
018     # If this line contains non-whitespace characters then return it.
019     if len(line.split()) > 0:
020       return line
021
022 # A Graphic as might appear in a _pages_ file.
023 class Graphic:
024   # Read a graphic string from _file_.
025   def __init__(self, file):
026     # _self.string_ is the entire multi-line graphic string.
027     self.string = ""
```

```
028     # A graphic must begin with a START command.
029     line = next_line(file)
030     if not line or not Graphic.is_start(line):
031       warn("Expected START at beginning of graphic.")
032       exit(1)
033     self.string += line
034     # For each START, an opposing END is required to close the graphic.
035     depth = 1
036     while True:
037       line = next_line(file)
038       # If we reached the end of file before the graphic is closed.
039       if not line:
040         warn("Graphic was not ended.")
041         exit(1)
042       # Add this line to the graphic string.
043       self.string += line
044       # If we encounter a START command then increase the depth.
045       # If we encounter an END command then decrease the depth.
046       # When depth reaches zero, the graphic is finished because an equal
047       # number of start and END commands have been read.
048       if Graphic.is_start(line):
049         depth += 1
050       elif Graphic.is_end(line):
051         depth -= 1
052         if depth == 0:
053           break
054   # Check if this line is a START graphic command.
055   def is_start(line):
056     # If the first field is START
057     fields = line.split()
058     if len(fields) < 1:
059       return False
060     return fields[0] == "START" or fields[0] == '"START"'
061   # Check if this line is an END graphic command.
062   def is_end(line):
063     # If the first field is END
064     fields = line.split()
065     if len(fields) < 1:
066       return False
067     return fields[0] == "END" or fields[0] == '"END"'
068
069 # Box and Glue are POLYMORPHIC classes, each are a type of gizmo.
070 # They both implement is_discardable, get_height, is_visible and print.
071 # Discardable gizmos found at the end of a flow are discarded.
072 # Visible gizmos will print a graphic.
073 class Box:
074   def __init__(self, height, graphic):
075     self.height = height
076     self.graphic = graphic
077   def is_discardable(self):
078     return False
079   def get_height(self):
080     return self.height
081   def is_visible(self):
082     return True
083   def print(self):
084     print(self.graphic.string, end='')
085 class Glue:
086   def __init__(self, height):
087     self.height = height
088   def is_discardable(self):
089     return True
090   def get_height(self):
091     return self.height
092   def is_visible(self):
093     return False
094   def print(self):
095     pass
096
097 # Given a list of gizmos, compute the height of this flow.
098 # Trailing discardable gizmos are discarded.
099 def gizmos_height(gizmos):
100   height = 0
101   # _discardable_height_ is the height of all discardable gizmos that are not
```

```
102   # followed by a non-discardable gizmo.
103   discardable_height = 0
104   for gizmo in gizmos:
105     if gizmo.is_discardable():
106       discardable_height += gizmo.get_height()
107     else:
108       # When a non-discardable gizmos is encountered, its height is added and
109       # any previous discardable gizmos are now also added because they are
110       # followed by a non-discardable gizmo.
111       height += discardable_height
112       height += gizmo.get_height()
113       discardable_height = 0
114   return height
115
116 # _PageGenerator_ is responsible for generating new Pages and assigning page
117 # numbers.
118 class PageGenerator:
119   def __init__(self, width, height, top_padding, bot_padding, left_padding,
120                right_padding, header_text):
121     self.width = width
122     self.height = height
123     self.top_padding = top_padding
124     self.bot_padding = bot_padding
125     self.left_padding = left_padding
126     self.right_padding = right_padding
127     self.header_text = header_text
128     self.page_count = 0
129   def new_page(self):
130     # Each time I make a new page, its page number is one more.
131     self.page_count += 1
132     return Page(self.width, self.height, self.top_padding, self.bot_padding,
133                 self.left_padding, self.right_padding, str(self.page_count),
134                 self.header_text)
135
136 # Stores the content of a single page.
137 class Page:
138   def __init__(self, width, height, top_padding, bot_padding, left_padding,
139                right_padding, page_number, header_text):
140     self.width = width
141     self.height = height
142     self.top_padding = top_padding
143     self.bot_padding = bot_padding
144     self.left_padding = left_padding
145     self.right_padding = right_padding
146     self.page_number = page_number
147     # _max_content_height_ is the height available for this page's content.
148     self.max_content_height = self.height - self.top_padding - self.bot_padding
149     self.header_text = header_text
150     # _normal_gizmos_ and _footnote_gizmos_ are each a 'flow'.
151     self.normal_gizmos = []
152     self.footnote_gizmos = []
153     # _marks_ is a list of mark strings that appear on this page.
154     self.marks = []
155     self.empty = True
156   def mark(self, mark):
157     # Mark this page with a string 'mark'. This will be added to the contents
158     # file which is used for generating a contents page.
159     self.marks.append(mark)
160   def write_marks(self, file):
161     # Marks are written to _contents_ files after all pages have been
162     # generated.
163     for mark in self.marks:
164       file.write('"{}" "{}"\n'.format(strip_string(mark), \
165           strip_string(self.page_number)))
166   # Force add gizmo content to this page.
167   # _normal_gizmos_ is a list of gizmos in the normal flow to append.
168   # _footnote_gizmos_ is a list of gizmos in the footnote flow to append.
169   def add_content(self, normal_gizmos, footnote_gizmos):
170     # If there is nothing to add then do nothing.
171     if len(normal_gizmos) == 0 and len(footnote_gizmos) == 0:
172       return
173     # The page is no-longer empty.
174     self.empty = False
175     # Append the gizmos to this page's flow.
```

```
176     self.normal_gizmos += normal_gizmos
177     self.footnote_gizmos += footnote_gizmos
178   # Try to add gizmo content. Only succeeds if there is sufficient space on
179   # this page.
180   # If there is not enough space for ALL gizmos then NO gizmos are added.
181   # Return True if successfully added.
182   def try_add_content(self, normal_gizmos, footnote_gizmos):
183     # _new_used_height_ is how tall the page content would be if the gizmos
184     # were added.
185     new_used_height = 0
186     # Add the height of the potential new normal and footnote flows.
187     new_used_height += gizmos_height(self.normal_gizmos + normal_gizmos)
188     new_used_height += gizmos_height(self.footnote_gizmos + footnote_gizmos)
189     # If the gizmos do not fit and the page is not empty then return False.
190     # If the gizmos do not fit but the page is empty then add them anyway
191     # because if they do not fit here then they will not fit anywhere and it
192     # would cause an infinite loop that keeps adding new empty pages and trying
193     # to fit the impossible content in each one.
194     if new_used_height > self.max_content_height and not self.empty:
195       return False
196     # Force add the new gizmos to this page.
197     self.add_content(normal_gizmos, footnote_gizmos)
198     return True
199   # Return the _pages_ graphic string for the page number.
200   def page_number_graphic(self):
201     # Build the _text_specification_ for the page number.
202     text = "FONT Regular 12\n"
203     text += 'STRING "{}"\n'.format(strip_string(self.page_number))
204     # line_break is called to centre the text.
205     graphic = line_break(text, \
206         self.width - self.left_padding - self.right_padding, 'c')
207     # Remove pager commands from the _content_ to turn it into a graphic that
208     # can be inserted into _pages_.
209     graphic = re.sub(r"opt_break.*", "", graphic)
210     graphic = re.sub(r"box.*", "", graphic)
211     return graphic
212   # Return the _pages_ graphic string for the page header.
213   def header_graphic(self):
214     # Build the _text_specification_ for the header.
215     text = "FONT Italic 12\n"
216     text += 'STRING "{}"\n'.format(strip_string(self.header_text))
217     # Break the header into lines.
218     graphic = line_break(text, \
219         self.width - self.left_padding - self.right_padding, 'c')
220     # Remove pager commands from the _content_ to turn it into a graphic that
221     # can be inserted into _pages_.
222     graphic = re.sub(r"opt_break.*", "", graphic)
223     graphic = re.sub(r"box.*", "", graphic)
224     return graphic
225   # Write this page's _pages_ content to standard output.
226   def print(self, show_page_number):
227     # Start the page.
228     print("START PAGE")
229     # Start at the top of the page minus the top padding.
230     y = self.height - self.top_padding
231     x = self.left_padding
232     # For each gizmo in the normal flow.
233     for gizmo in self.normal_gizmos:
234       # Move down the page by this gizmo's height.
235       y -= gizmo.get_height()
236       # If the gizmo is visible then move to (x, y) and print it.
237       if gizmo.is_visible():
238         print("MOVE {} {}".format(x, y))
239         gizmo.print()
240     # Now set _y_ to the bottom of the page plus the bottom padding plus the
241     # total footnote flow height.
242     y = self.bot_padding + gizmos_height(self.footnote_gizmos)
243     # For each gizmo in the footnote flow.
244     for gizmo in self.footnote_gizmos:
245       # Move down by this gizmos height.
246       y -= gizmo.get_height()
247       # If the gizmo is visible then move to (x, y) and print it.
248       if gizmo.is_visible():
249         print("MOVE {} {}".format(x, y))
```

```
250         gizmo.print()
251     # If page numbers are enabled then move to the bottom of the page and write
252     # the page number graphic.
253     if show_page_number:
254       print("MOVE {} {}".format(self.left_padding, self.bot_padding // 2))
255       print(self.page_number_graphic())
256     # If the header is not empty then move to the top of the page and print it.
257     if self.header_text != "":
258       print("MOVE {} {}".format(self.left_padding, self.height - self.top_padding // 2))
259       print(self.header_graphic())
260     # End the page.
261     print("END")
262
263 # Parse pager command line arguments.
264 arg_parser = argparse.ArgumentParser()
265 arg_parser.add_argument("-l", "--left_margin", type=int, default=102)
266 arg_parser.add_argument("-r", "--right_margin", type=int, default=102)
267 arg_parser.add_argument("-t", "--top_margin", type=int, default=125)
268 arg_parser.add_argument("-b", "--bot_margin", type=int, default=125)
269 # -c / --contents Where to output the contents file. Default is None.
270 arg_parser.add_argument("-c", "--contents")
271 # If the -n flag is present then page numbers will be drawn.
272 arg_parser.add_argument("-n", "--page_numbers", action="store_true")
273 arg_parser.add_argument("-H", "--header", default="")
274 args = arg_parser.parse_args()
275
276 # 595x842 is the point resolution of an A4 page.
277 page_generator = PageGenerator(595, 842, args.top_margin, args.bot_margin, \
278     args.left_margin, args.right_margin, args.header)
279 pages = []
280 # Generate an initial page.
281 active_page = page_generator.new_page()
282 # Pending gizmos are stored in a buffer. When an (optional) page break is read,
283 # they are removed from this buffer and added to a Page object.
284 pending_gizmos = {"normal": [], "footnote": []}
285 current_flow = "normal"
286 # For each record parsed in this programs standard input.
287 while fields := parse_record(sys.stdin):
288   # The first field in the record is the pager command type.
289   if fields[0] == "flow":
290     # flow [normal/footnote]
291     if len(fields) != 2:
292       warn("flow command expects one argument.")
293       continue # Go to the next record.
294     # If the argument of the flow command is not "normal" or "footnote":
295     if not fields[1] in pending_gizmos.keys():
296       warn("invalid flow '{}'".format(fields[1]))
297       continue
298     # Update the _current_flow_ to the argument of this flow command.
299     current_flow = fields[1]
300   elif fields[0] == "mark":
301     # mark MARK_STRING
302     if len(fields) != 2:
303       warn("mark command expects one argument.")
304       continue
305     # Mark this page with the string argument of the mark command.
306     active_page.mark(fields[1])
307   elif fields[0] == "box":
308     # box GRAPHIC_HEIGHT
309     # [pages graphic]
310     if len(fields) != 2:
311       warn("box command expects one argument.")
312       continue
313     # Try to convert the argument of the command to an integer. If it does not
314     # work then just default to a height of zero.
315     try:
316       height = int(fields[1])
317     except:
318       warn("box command argument must be integer.")
319       height = 0
320     # Box command is followed by a pages graphics so read this graphic and
321     # store it in _graphic_.
322     graphic = Graphic(sys.stdin)
323     # Add this Box to the pending gizmos in the current flow.
```

```
324     box = Box(height, graphic)
325     pending_gizmos[current_flow].append(box)
326   elif fields[0] == "glue":
327     # glue GLUE_HEIGHT
328     if len(fields) != 2:
329       warn("glue command expects one argument.")
330       continue
331     # Try to convert the argument of the command to an integer. If it does not
332     # work then just default to a height of zero.
333     try:
334       height = int(fields[1])
335     except:
336       warn("glue command argument must be integer.")
337       height = 0
338     # Add this glue to the pending gizmos of the current flow.
339     glue = Glue(height)
340     pending_gizmos[current_flow].append(glue)
341   elif fields[0] == "opt_break" or fields[0] == "new_page":
342     # opt_break and new_page have no arguments.
343     # They both require flushing the pending gizmos.
344     # If the pending gizmos wont fit on the current page:
345     if not active_page.try_add_content(pending_gizmos["normal"],
346                                        pending_gizmos["footnote"]):
347       # Then make a new page and add the gizmos onto that.
348       pages.append(active_page)
349       active_page = page_generator.new_page()
350       active_page.add_content(pending_gizmos["normal"],
351                               pending_gizmos["footnote"])
352     # The pending gizmos have now been flushed so _pending_gizmos_ is reset.
353     pending_gizmos = {"normal": [], "footnote": []}
354     # If we wanted a new page and the current page is not empty then make a new
355     # page.
356     if fields[0] == "new_page" and not active_page.empty:
357       pages.append(active_page)
358       active_page = page_generator.new_page()
359   else:
360     # Good example of error handling.
361     warn("unrecognised command '{}'".format(fields[0]))
362
363 # Flush left-over pending gizmos.
364 if not active_page.try_add_content(pending_gizmos["normal"],
365                                    pending_gizmos["footnote"]):
366   pages.append(active_page)
367   active_page = page_generator.new_page()
368   active_page.add_content(pending_gizmos["normal"],
369                           pending_gizmos["footnote"])
370 pages.append(active_page)
371
372 # If the user wanted a contents file then make it.
373 contents = None
374 if args.contents:
375   contents = open(args.contents, "w")
376 # Loop over all pages.
377 for page in pages:
378   # Print the _pages_ content to standard output.
379   page.print(args.page_numbers)
380   # If we are using a contents file then write this page's marks to it.
381   if contents:
382     page.write_marks(contents)
383 # If we used a contents file then close it.
384 if contents:
385   contents.close()
```

4 Testing

Testing in the early stages of my project was achieved with the 'doc.sh' shell script. This script used the project's executables to generate a PDF that demonstrated all the features of current system. After each new change change, I could verify the program worked by running the shell script and inspecting the PDF. The script was updated to use new features as they were added.

The following code listing shows the latest version of 'doc.sh'. It includes a large amount of *lorem ipsum* (Latin sample text that has been traditionally used to test text-processing systems). It also has a contents page, an image and a custom page inserted at the end of the document. Footnotes with a very large font size are used to demonstrate that they will never overlap with the main page content.

```
#!/bin/sh

PATH=$PATH:..

(sh | pager.py -n -c .contents) > .content_pages << END_CONTENT

echo 'mark "document start"'

(markup_text.py -w 390 -A r -S 20 -P 50 -L 10) << END_PARAGRAPH
Hello _world,_ this is a footnote
^1 this is the content of the first footnote which spans multiple lines
and some more text
^2 this is a second footnote
here which continues until a line break occurs.

some text here
# Header 1
## Header 2
### Header 3
#### Header 4

A new *paragraph begins here.
END_PARAGRAPH

echo 'mark "before graphic"'

echo "glue 10"
echo "box 100"
echo "START GRAPHIC"
echo "IMAGE 100 100 peppers.jpg"
echo "END"
echo "glue 10"

echo 'mark "after graphic"'

(markup_text.py -w 390 -s 10 -a j -p 20 -l 2) << END_PARAGRAPH
Lorem* ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Cursus sit amet dictum sit amet
justo donec enim diam. Amet luctus venenatis lectus magna fringilla. Sit amet
purus gravida quis. Mollis aliquam ut porttitor leo a diam sollicitudin tempor.
Leo a diam sollicitudin tempor. Vitae ultricies leo integer malesuada nunc vel
```

risus commodo viverra. Mollis aliquam ut porttitor leo. Nunc pulvinar sapien et ligula ullamcorper malesuada proin libero nunc. Eu augue ut lectus arcu bibendum at varius vel. Eget gravida cum sociis natoque penatibus et magnis dis. In tellus integer feugiat scelerisque varius morbi. Ullamcorper sit amet risus nullam eget felis. Tortor dignissim convallis aenean et tortor.

Posuere ac ut consequat semper viverra. Magna fringilla urna porttitor rhoncus dolor purus non. Faucibus pulvinar elementum integer enim neque volutpat ac. Nunc mi ipsum faucibus vitae aliquet nec ullamcorper. Felis bibendum ut tristique et egestas quis. Habitasse platea dictumst quisque sagittis. Non enim praesent elementum facilisis leo vel fringilla est ullamcorper. Rhoncus mattis rhoncus urna neque viverra justo nec ultrices dui. Cursus vitae congue mauris rhoncus aenean. Urna nec tincidunt praesent semper feugiat nibh sed pulvinar. Faucibus ornare suspendisse sed nisi lacus sed viverra tellus in. In hac habitasse platea dictumst quisque sagittis purus. Tellus integer feugiat scelerisque varius. Tellus integer feugiat scelerisque varius morbi. Vitae ultricies leo integer malesuada nunc vel risus commodo viverra. Commodo quis imperdiet massa tincidunt nunc pulvinar sapien et. Enim facilisis gravida neque convallis a cras semper auctor. Aenean vel elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi. Orci eu lobortis elementum nibh tellus. Cras adipiscing enim eu turpis egestas.

Euismod elementum nisi quis eleifend quam. Felis imperdiet proin fermentum leo. Id interdum velit laoreet id donec ultrices tincidunt arcu non. At imperdiet dui accumsan sit amet nulla. Arcu cursus euismod quis viverra nibh cras pulvinar mattis nunc. Turpis tincidunt id aliquet risus feugiat. Aliquet sagittis id consectetur purus ut faucibus pulvinar elementum. Risus commodo viverra maecenas accumsan. Consectetur purus ut faucibus pulvinar elementum integer enim neque. Pharetra diam sit amet nisl suscipit adipiscing. Lacus laoreet non curabitur gravida arcu. Sit amet tellus cras adipiscing enim eu. Molestie ac feugiat sed lectus vestibulum mattis ullamcorper velit. Fames ac turpis egestas sed tempus. Elementum nibh tellus molestie nunc non blandit. Suscipit adipiscing bibendum est ultricies integer. Pellentesque habitant morbi tristique senectus et netus.

Libero volutpat sed cras ornare arcu dui vivamus arcu felis. Ut faucibus pulvinar elementum integer enim neque volutpat ac. Nec ullamcorper sit amet risus nullam eget. A iaculis at erat pellentesque. Nascetur ridiculus mus mauris vitae ultricies leo integer. Vitae ultricies leo integer malesuada nunc vel risus. Aliquam malesuada bibendum arcu vitae elementum curabitur. Rhoncus est pellentesque elit ullamcorper dignissim cras. Nisi est sit amet facilisis magna etiam tempor orci eu. Mauris pharetra et ultrices neque ornare aenean. Ac tincidunt vitae semper quis. Gravida quis blandit turpis cursus in hac. Egestas congue quisque egestas diam in arcu. Id aliquet risus feugiat in. Dui nunc mattis enim ut tellus elementum sagittis. Id interdum velit laoreet id donec ultrices tincidunt arcu. Neque viverra justo nec ultrices dui sapien eget mi. Sit amet purus gravida quis blandit turpis cursus. Proin nibh nisl condimentum id.

Sit amet nisl suscipit adipiscing bibendum est. Posuere urna nec tincidunt praesent semper feugiat. At erat pellentesque adipiscing commodo elit. Fermentum dui faucibus in ornare quam viverra. Ipsum nunc aliquet bibendum enim facilisis gravida neque. Odio ut enim blandit volutpat maecenas volutpat blandit aliquam. Aliquet eget sit amet tellus cras adipiscing enim eu turpis. Id consectetur purus ut faucibus pulvinar elementum integer. Eu mi bibendum neque egestas congue quisque egestas diam in. Turpis nunc eget lorem dolor sed viverra. Elit duis tristique sollicitudin nibh sit amet commodo. Sed nisi lacus sed viverra. Eget magna fermentum iaculis eu non diam. Gravida in fermentum et sollicitudin ac. Facilisi nullam vehicula ipsum a. Malesuada proin libero nunc consequat interdum varius sit amet mattis.

More content that spills to the next page

END_PARAGRAPH

```
echo 'mark "end content"'

END_CONTENT

(sh | tw) << END_DOCUMENT
(contents.py | pager.py) < .contents
cat .content_pages
cat << END_PAGE
START PAGE
MOVE 100 420
START TEXT
FONT Regular 12
STRING "END OF DOCUMENT"
END
END
END_PAGE
END_DOCUMENT
```

After my project reached a sufficient set of features, I wrote a shell script to typeset and generate a PDF of the most recent draft of the NEA report. All subsequent report drafts (including this final one) were typeset with this project's software. Practical use of the software helped me uncover bugs and identify important new features.

In order to test the output document's compliance with the 1.7 PDF standard, I used a web based PDF validation tool * throughout the project's development.

I published this * unlisted youtube video that demonstrated all objectives. The video shows the compilation of 3 documents, one for each objective category. In each document, text is labled with the objective number that is demonstrated.

The following shell script generates *content* for the *pager* program. Below the shell script source, a page is included with content generated by the script. This page is designed to demonstrate and text the capability of the software.

```
#!/bin/sh

markup_text.py -w 390 << END_TEXT
## Objectives Met Demonstration

# Large Header
## Small Header

Hello world, this is a demonstration! *Bold text,* _italic text,_ a line break
occurs here.
Footnotes
^* footnotes are indeed supported.
and images are supported:
END_TEXT

echo "box 110"
echo "START GRAPHIC"
echo "IMAGE 100 100 peppers.jpg"
echo "END"

line_break -w 100 -r << END_FONT_DEMO
FONT Regular 12
STRING "Other"
OPTBREAK " " "" 0
FONT Demo 12
STRING "fonts"
```

* *https://www.pdf-online.com/osa/validate.aspx*

* *https://youtu.be/58OCfdlV1o0*

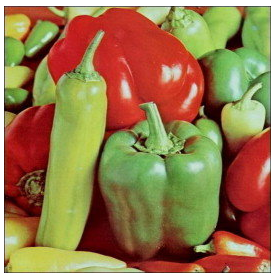
```
OPTBREAK " " "" 0
FONT Regular 12
STRING "and"
OPTBREAK " " "" 0
FONT Regular 18
STRING "sizes"
FONT Regular 12
OPTBREAK " " "" 0
STRING "can"
OPTBREAK " " "" 0
STRING "be"
OPTBREAK " " "" 0
STRING "mixed."
END_FONT_DEMO

echo "glue 10"

line_break -w 130 << END_LINE_DEMO
FONT Regular 12
STRING "This"
OPTBREAK " " "" 0
STRING "text"
OPTBREAK " " "" 0
STRING "demonstrates"
OPTBREAK " " "" 0
STRING "line"
OPTBREAK " " "" 0
STRING "breaks"
OPTBREAK " " "" 0
STRING "and"
OPTBREAK " " "" 0
STRING "hyphen"
OPTBREAK "" "-" 0
STRING "ation."
END_LINE_DEMO

cal -y | markup_raw.py -s 6
```

Objectives Met Demonstration

Large Header

Small Header

Hello world, this is a demonstration! **Bold text,** *italic text,* a line break occurs here. Footnotes * and images are supported:



Other fonts and sizes can be mixed.

This text demonstrates line breaks and hyphenation.

2023

```
      January               February              March
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
                   1            1  2  3  4  5         1  2  3  4  5
 2  3  4  5  6  7  8   6  7  8  9 10 11 12   6  7  8  9 10 11 12
 9 10 11 12 13 14 15  13 14 15 16 17 18 19  13 14 15 16 17 18 19
16 17 18 19 20 21 22  20 21 22 23 24 25 26  20 21 22 23 24 25 26
23 24 25 26 27 28 29  27 28                 27 28 29 30 31
30 31
      April                 May                   June
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
                1  2   1  2  3  4  5  6  7            1  2  3  4
 3  4  5  6  7  8  9   8  9 10 11 12 13 14   5  6  7  8  9 10 11
10 11 12 13 14 15 16  15 16 17 18 19 20 21  12 13 14 15 16 17 18
17 18 19 20 21 22 23  22 23 24 25 26 27 28  19 20 21 22 23 24 25
24 25 26 27 28 29 30  29 30 31              26 27 28 29 30

      July                  August                September
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
                1  2      1  2  3  4  5  6               1  2  3
 3  4  5  6  7  8  9   7  8  9 10 11 12 13   4  5  6  7  8  9 10
10 11 12 13 14 15 16  14 15 16 17 18 19 20  11 12 13 14 15 16 17
17 18 19 20 21 22 23  21 22 23 24 25 26 27  18 19 20 21 22 23 24
24 25 26 27 28 29 30  28 29 30 31           25 26 27 28 29 30
31
      October               November              December
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
                   1            1  2  3  4  5               1  2  3
 2  3  4  5  6  7  8   6  7  8  9 10 11 12   4  5  6  7  8  9 10
 9 10 11 12 13 14 15  13 14 15 16 17 18 19  11 12 13 14 15 16 17
16 17 18 19 20 21 22  20 21 22 23 24 25 26  18 19 20 21 22 23 24
23 24 25 26 27 28 29  27 28 29 30           25 26 27 28 29 30 31
30 31
```

* *footnotes are indeed supported.*

5 Evaluation

5.1 Requirements Met

The project has met it's initial requirements as defined by the project statement and list of objectives. My experience in typesetting this report with the software has been good; it is clear to me that I have produced an effective and useful tool.

5.2 Improvements

In its current state, the software considers line breaking and page breaking to be two separate problems with a separate implantation. This simplifies each individual problem but also limits the software capability to solve more complex problems in which line breaking and page breaking can not be considered separately (perhaps the width of text is constrained by where it falls on a page). I envision a so called 'universal content breaker' which is given a complete model of all content to be typeset and a well defined objective function. This universal content breaker will handle both line breaks, page breaks and any other type of content separation that is required in a single optimization problem (most likely modeled by a shortest path problem as the line breaking currently is). By centralizing the typesetting problem, the document content and typesetting objective can be defined more rigorously without being limited by pre-selected line breaks (as the pager currently is). This would also reduce the number of file formats that the user must know, potentially making the product easier to use.

5.3 Feedback

I spoke to the third-party of the project - Anthony Ceponis - about the finished project. We looked at the shell script that builds the report and discussed how my new typesetting system compares to alternatives such as LaTeX. The following is an extract of Anthony's comments.

I definitely think that this is a huge improvement over LaTeX. The main reason I prefer this new system you have built is because much less syntax is needed to achieve the exact same things without jeopardising the clarity of the markup. On the note of the markup itself, I really like how human readable it is. For example, there is a very limited use of non alpha numeric symbols (like angle brackets which are abused in html) which I think is a huge plus.

I would not really compare this to something like Google docs because I think they are both built for different things. Google docs is very much a 'user' product rather than a developer oriented product which is why I don't think it would be that appropriate to compare them. I can easily see your typesetting system being used by people like web-developers to create complex documents with complex structures or even for simpler use like storing structured/formatted blogs on a database.

Obviously this project is still very young so my suggested improvements would have probably been incorporated over time anyways but I would be interested to see how things like mathematical symbols would be represented (e.g. integral and sigma signs) in the markup because currently I use latex for this and I am not a huge fan of how it works currently. I also dislike the bracketing system in latex (the height

of brackets should be able to adjust automatically without the need for extra code). Some more complicated features like tables would also be interesting to see. Clearly the only improvements I have are to just add more features but as of now, I don't have any complaints about changing any existing markup systems in your project.

I agree with Anthony's remark that the project is more suitable for developers than for the average computer user. Perhaps an optional, additional layer of abstraction could be provided to simplify basic usage of the system for novice Unix users.

I disagree however, with Anthony's belief that the project simply needs to 'just add more features'. I believe that the best way to improve on the project is to redesign the fundamental system to give the user a set of simple cohesive tools which can be combined to produce more complex features such as tables.

Overall, I think Anthony's comments reflect my assertion that the project fulfills its original goals, though it could be improved further in some ways if more time was available.